UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

PAULO RICARDO RODRIGUES DE SOUZA JUNIOR

A data-driven dispatcher for Big Data applications in Heterogeneous Systems

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Advisor: Prof. Dr. Claudio Resin Geyer

Porto Alegre
Setembro 2018

CIP — CATALOGING-IN-PUBLICATION

Rodrigues de Souza Junior, Paulo Ricardo

A data-driven dispatcher for Big Data applications in Heterogeneous Systems / Paulo Ricardo Rodrigues de Souza Junior. – Porto Alegre: PPGC da UFRGS, 2018.

104 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2018. Advisor: Claudio Resin Geyer.

1. Big data. 2. Load balance. 3. Data-stream partition. 4. Communication optimization. I. Geyer, Claudio Resin. II. Título.

“The best way to predict the future is to to create it.”

— PETER DRUCKER

ABSTRACT

Mankind is increasing technology capacity every day, as it is taking place in multiple areas like automation, predicting, making actions, and so on. In this process, data is produced in different ratios and quantities, and from a close point of view the data production of a single sensor is not much and does not provide clear insights. However, a global vision and the union of that information may contain helpful knowledge about business intelligence, people and sensor behavior. The global view of all this data is called Big Data and may achieve overwhelming amounts of data, which is being produced in outstanding rates by devices and people. Therefore, it is necessary to provide solutions to manage Big Data systems, which give robustness and quality of service. In order to achieve robust systems to process high amounts of data, Big Data frameworks are proposed and deployed using several management tools. Furthermore, Big Data frameworks are usually separated in different perspectives of processing (i.e., batch and stream processing), and focuses on processing balanced data in homogeneous environments. Stream and Batch Processing Engines have to support high data ingestion to ensure the quality and efficiency for the end-user or a system administrator. The data flow processed by SPE fluctuates over time and requires real-time or near real-time resource pool adjustments (network, memory, CPU and other). This scenario leads to the problem known as skewed data production caused by the non-uniform incoming flow at specific points on the environment, resulting in slow down of applications produced by network bottlenecks and inefficient load balance. The current proposal of this thesis is the Aten a data-driven dispatcher as a solution to overcome unbalanced data flows processed by Big Data Stream applications in heterogeneous systems. Aten manages data aggregation and data streams within message queues, assuming different algorithms as strategies to partition data flow over all the available computational resources. The thesis presents results indicating that is possible to maximize the throughput and also provide low latency levels for SPEs.

Keywords: Big data. load balance. data-stream partition. communication optimization.

Um dispatcher acionado por dados de aplicações de Big Data em sistemas heterogêneos

RESUMO

A tecnologia vem se desenvolvendo cada vez mais, permeando diversas áreas como automação, predição, tomada de decisões e etc. Neste processo, os dados são produzidos a taxas e quantidades diferentes, e inicialmente não significam muito. Entretanto, a partir de uma visão geral e união dos dados podem gerar informações úteis sobre *Business Inteligence* (BI), a população e o comportamento de aplicações. Esse ponto de vista abrangente também é chamado de Big Data podendo englobar gigantescas massas de dados, que são produzidos rapidamente por dispositivos e pessoas. Assim, se faz necessário fornecer soluções para gerenciar os sistemas de Big Data, dada sua robustez e a qualidade requerida. Os sistemas Big Data processam grandes volumes de dados, assim os *frameworks Big Data* são desenvolvidos e implementados utilizando diversas ferramentas de gerenciamento. Estes *frameworks* são geralmente divididos em duas perspectivas de processamento diferentes (i.e., *batch* e *stream processing*), que consideram o processamento de dados distribuídos em ambientes homogêneos. *Stream* e *Batch processing engines* devem suportar uma grande entrada de dados para manter a qualidade e eficiência para o cliente final ou para o administrador do sistema. A taxa do fluxo de dados processados pelos SPS pode variar durante o tempo, exigindo que os recursos computacionais (rede, memória, CPU e outros) sejam ajustados em tempo real ou próximo do real. Esse cenário acaba produzindo dados distorcidos originados pela desuniformidade do fluxo de dados em determinados pontos do sistema, resultando na lentidão das aplicações provocada pelos gargalos de rede e pelo desbalanço de carga. O principal objetivo desta tese é propor o Aten, um *dispatcher* orientado a dados, como solução para o fluxo desbalanceado dos dados nas aplicações de Stream Big Data em ambientes heterogêneos. Aten coordena a agregação e o fluxo de dados durante a sua passagem pelas filas de mensagem, assumindo diferentes algoritmos como estratégias para particionar os dados entre todos os recursos computacionais disponíveis. Neste trabalho também são apresentados os resultados que indicam que é possível maximizar o *throughput* e diminuir a latência dos sistemas de processamento de dados.

Palavras-chave: Big Data, Balanceamento de Carga, Fluxo de Dados, Comunicaçção Otimizada.

LIST OF ABBREVIATIONS AND ACRONYMS

AG	Acyclic Graph
AMQP	Advanced Message Queuing Protocol
BDA	Big Data Analytics
DAG	Directed Acyclic Graph
DSMS	Data Stream Management System
DSPE	Distributed Stream Processing Engines
ETL	Extract Transform Load
ESP	Event-Stream Processing
HDFS	Hadoop Distributed File System
JMS	Java Message Service
MR	Map Reduce
RT	Real Time
RDD	Resilient Distributed Dataset
RPC	Remote Procedure Call
SPE	Stream Processing Engines
SPS	Stream Processing System
STOMP	Streaming Text Oriented Messaging Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

LIST OF FIGURES

Figure 2.1 Big Data: The 5 Vs. Source: (MATTEUSSI et al., 2017) 18
Figure 2.2 MapReduce Parallel Execution. Source: (DEAN; GHEMAWAT, 2009)......21
Figure 2.3 Stream Processing Concept. Source: the author...23
Figure 2.4 Lambda Architecture. Source: Rewritten by the author (MARZ; WARREN, 2015) 26
Figure 2.5 Kappa Architecture. Source: (KAPPA. . . , 2014) 28

Figure 3.1 Apex Architecture. Source: (PATHAK; RATHI; PAREKH, 2016)...............33
Figure 3.2 Spark Architecture. Source: (INC, 2016) 35
Figure 3.3 Flink Architecture. Source: (JAVED; LU; PANDA, 2017) 37
Figure 3.4 Kafka Architecture. Source: (KREPS, 2011) 40

Figure 4.1 Liquid Stack (FERNANDEZ et al., 2015)..46
Figure 4.2 Liquid model (FERNANDEZ et al., 2015)..47
Figure 4.3 Cirus Architecture (PHAM et al., 2016)...48
Figure 4.4 SMART Architecture (ANJOS et al., 2015)...51

Figure 5.1 Aten Model. Source: the author..60
Figure 5.2 Data Orchestrator Flow. Source: the author..61
Figure 5.3 Event average duration in ms...63
Figure 5.4 Stream Partition. By the author...65
Figure 5.5 Aten Prototype. By the author 67

Figure 6.1 Experimental Platform..70
Figure 6.2 Single event duration in ms 73
Figure 6.3 Single event duration in ms 74
Figure 6.4 Throughput per node 75
Figure 6.5 Average Throughput per algorithm...76
Figure 6.6 All scenarios - Throughput per node 77
Figure 6.7 Average CPU Usage per node 79
Figure 6.8 Average Memory Usage per node 79
Figure 6.9 Average Network Receive per node..80
Figure 6.10 Average Network Send per node 81

Figure A.1 Stream Processing Flow. Designed by the author..94
Figure A.2 Execution time over the entire model 96
Figure A.3 Event time..96
Figure A.4 Time by event...97

LIST OF TABLES

Table 3.1 Comparative Between Big Data Frameworks 42
Table 3.2 Comparative Between Message Queues Frameworks....................................44

Table 4.1 Related Work Comparative 57

Table 6.1 Microsoft Azure Instances Details 70
Table 6.2 Total duration time in seconds....................................78
Table 6.3 Average Execution time comparison....................................82

CONTENTS

1 INTRODUCTION 11
1.1 Motivation 13
1.2 Problem and Contribution 14
1.3 Goals 15
1.4 Methodology 16
1.5 Organization 16
2 BIG DATA BACKGROUND 18
2.1 Big Data Overview 18
2.2 MapReduce Programming Model 20
2.3 Big Data Processing Models 22
2.3.1 Batch Processing 22
2.3.2 Stream Processing 23
2.4 Big Data Architectures Overview 25
2.4.1 Lambda Architecture 25
2.4.2 Kappa Architecture 27
2.5 Discussion 28
3 STREAM PROCESSING ENGINES 30
3.1 Overview 30
3.2 Big Data Platforms 31
3.2.1 Hadoop Streaming 31
3.2.2 Apache Apex 32
3.2.3 Apache Storm 34
3.2.4 Apache Spark 34
3.2.5 Apache Flink 36
3.2.6 Apache Samza 37
3.3 Message Queues 38
3.4 Discussion 41
4 RELATED WORK 45
4.1 Dispatcher Models in proposals 45
4.1.1 Liquid 45
4.1.2 Cirus 48
4.1.3 JetStream 49
4.1.4 SMART 50
4.1.5 Neptune 51
4.2 Communication Optimization 52
4.3 Stream Partition 55
4.4 Discussion 56
5 ATEN 59
5.1 Proposal Overview 59
5.2 Dispatcher Model 60
5.2.1 Ingestion Module 60
5.2.2 Controller Module 64
5.3 Prototype 67
6 EVALUATION AND RESULTS 69
6.1 Environment 69
6.2 Methodology 71
6.3 Results 72
6.3.1 Events Duration 72

6.3.2 Events Throughput 74
6.3.3 Resource Consumption 78
6.4 Discussion 81
7 FINAL CONSIDERATIONS 83
7.1 Conclusions 83
7.2 Future Work 84
REFERENCES 86
APPENDIX A — BUFFERING INITIAL EXPERIMENTS 94
A.0.1 Results 95
APPENDIX B — CAPíTULO DE RESUMO EM PORTUGUÊS 98
B.1 Introdução 98
B.2 Background em Big Data 98
B.3 Trabalhos Relacionados 100
B.4 Aten: modelo proposto 102
B.5 Avaliação e resultados 103
B.6 Conclusões 103

1 INTRODUCTION

Organizations and companies extract information from huge volumes of data that are produced by several devices or clients, and this is possible since Big Data enabled the analysis of massive amounts of data. Although, there is still loss of data, without extracting potential knowledge for lack of means of mining the information or not having the necessary resources to process it. In order to enable the extraction of that knowledge, it is necessary a vast capacity of computational resources.

Internet of Things (IoT) has a role in the data production process, because of the possibility of collecting data from the most varied devices. The capacity to store and process huge amounts of data has been possible with the adoption by Big Data of the MapReduce model introduced in 2004 by (GHEMAWAT, 2004) which enabled the capacity of analyzing huge amounts of data. It has evolved constantly and now it is being divided into two main subclasses that are not only based on MapReduce: Stream Processing and Batch Processing. Batch Processing is the manipulation of intrinsically large data (usually already available for analysis), thus not requiring real-time data. Since Batch-processing was not structured to have low latencies, a new model was introduced, the Stream-Processing model, which deals with quite small batches. These specific tiny data (often denoted as events) are usually characterized by a small unit size (in the order of kilobytes) that has overwhelming collection rates (MATTEUSSI et al., 2017). As examples may be cited corporations like Google and Facebook, responsible for generating and maintaining large amounts of data from their users (POLATO et al., 2014). In recent years, the subclass of Big Data called fast data (MILOSLAVSKAYA; TOLSTOY, 2016) (i.e., high-speed real-time and near-real-time data streams, Stream-Processing) has witnessed a vast increase in volume and availability.

According to (TUDORAN; COSTAN; ANTONIU, 2016), Stream-Processing has been applied in multi-cloud environments and brought scientific discoveries. Moreover, stream processing allows to performing most of the processing independently on the data partitions across different sites and then to aggregating the results. In some of the largest scenarios, the data sets are already partitioned for storage across multiple sites, which simplifies the task of preparing and launching a geographical distributed processing. Multi-Cloud (MC) has characteristics as heterogeneity and variety of network bandwidth. The use of MC is inevitable since a single site will not be enough to handle the high volume of data (TUDORAN et al., 2014a).

However, the recent ascension of the stream processing does not overcome the necessity of batch processing. Still, there are requirements of cross processing, i.e., the hybrid processing, when it is necessary to cross the data from stream events against the historical data, constituted by big batches. Therefore, frameworks for the processing of stream and batch typically require environments that are capable of supporting large-scale data processing, structured and unstructured data, high data ingestion rates, and resource availability variations. Nonetheless, recent studies (ABAWAJY, 2015; XAVIER et al., 2015; XAVIER et al., 2016) met unexpected network contention problems (e.g., low throughput and high end-to-end latency) and poor distribution of data load as the main issues while processing Big Data Streaming applications in Cloud environment.

Cloud Computing has emerged as a scalable infrastructure, which provides an extensive number of geographically distributed data centers, which is ideal for Big Data Analytics. Frameworks for hybrid processing applications typically require environments that are capable of supporting large-scale data processing, structured and unstructured data, high data ingestion rates, and resource availability variations. Under these circumstances, cloud computing enables resource sharing, scalability, and resiliency to address these challenges.

Thus, in this work, it is proposed Aten, a dispatcher to distribute tasks in the hybrid processing engines. Managing the stream and batch data, applying techniques over the communication and load distribution in order to overcome the limitations that are still found in the state-of-the-art. Aten aims at the implementation of solutions for hybrid processing, focusing mainly in Stream processing constraints - such as imbalanced load distribution and resource management caused by skewed queues and, consequently, low throughput of events - by proposing and evaluating multiple methods of stream partitioning and communication optimization, in order to increase the throughput of events and exploitation of resources (e.g., CPU, Memory).

Aten is built using popular Big Data tools, promoting a high-level model that can be adapted to individual needs. Thus, the evaluated algorithms must self-adapt to every particularity present in these systems. It happens mainly because each system implements the communication between operators differently. Therefore, an in-depth study of the related work is conducted and benchmarked alongside the proposal in order to achieve crucial contributions. Besides, a prototype is developed from the Aten model, and it is evaluated and analyzed to validate the proposal.

1.1 Motivation

The existing gap between the growth rate of the processing power and data storage speed is a historical fact known for more than ten years. However, recently this gap does not influence applications that are data intensive, either they be from the IoT field, social networks, the supply chain or the industry. These applications are strongly linked to the ability to simultaneously store and process large amounts of data (i.e. Big Data), so that information can be extracted from it to advantage decision making (ASSUNÇÃO et al., 2015).

The recent scenario of Big Data applications has important requirements in the design of applications: it is essential to ensure the quality of the results obtained from the information provided; it is required for a result extraction process, whether it be an indicator or a solution metric, meet the application deadline; it is necessary to establish a robust and scalable solution that provides self-adaptability from changes and variations in the data flow (WHITE, 2012). Given this, it is possible to find a series of solutions and proposals in state of the art to solve the challenges in meeting these requirements.

Some varieties of proposals are found in state of the art, with specific suggestions that do not always cover the global scenario and only cater to particular situations. For instance, proposals that focus on reducing latency in stream processing systems, somehow lack on partitioning the data stream inside an environment with heterogeneous resources and vice versa. Even though a proposal not necessary need to attend every problem within a system, the difficulties cited in the next section are strongly correlated and should be taken into account together to achieve a great global result. Therefore, this research sees as an opportunity to be explored in order to overcome constraints of the state of the art and propose a singular solution that somehow will provide further advancements in that research area.

Furthermore, stream processing events are easily ingested by the system and are picked up by multiple and different devices, thus generating large income ratio over input streams that requires little response time. Due to these characteristics, the processing of these events occurs in memory (Apache Storm (TOSHNIWAL et al., 2014), Apache Spark (INC, 2016), S4 (NEUMEYER et al., 2010), Apache Samza (ZHUANG et al., 2016), Apache Flink (ALEXANDROV et al., 2014), and so on), since accessing by writing and reading on hard disk delays operations. Of course, there is a need to process information using both of the subclasses, as introduced by the Lambda architecture. Lambda Architectures are designed to handle vast amounts of data in conjunction with both *batch* and *stream* processing methods (SILVA et al., 2016).

1.2 Problem and Contribution

The data-intensive applications have broad requirements, which can be factored into many difficulties and problems that require attention. The most common problems encountered on stream processing systems which run in heterogeneous environments are discussed in this section.

In stream processing applications, from their starting point (i.e., the data production), significant weaknesses are found. For example, once a device produces the data it must be sent to a processing location, during this process some limitations are the definition of a proper route for that data to be delivered; failures during data serialization of the message; delayed or even lost messages, which may require re-submission (JUNIOR et al., 2018).

Nevertheless, during the reception of messages, they must be stored and prepared for consumption in the processing environment. The Message Brokers are responsible for persisting and pipelining of these messages (JAVED; LU; PANDA, 2017). It is necessary to establish storage and memory management approaches to manipulate a high flow of events, whether these messages are small or large. Besides, some points directly influence the performance of the application (KLEPPMANN; KREPS, 2015): the number of replicas defined in the message queues; the number of consumers and the partitioning of these message queues. The distribution of these messages to be processed in the processing engine is a vital part of the process of executing an application. Different forms of partitioning and data placement(NASIR et al., 2015), or even total absence, can cause unbalance in the division of workload, thus causing resource contention and under utilization (MATTEUSSI; ROSE, 2015). The waste or misuse of available resources in a data-processing cluster can outcome in substantial increases in runtime, decreased throughput, not meeting the real-time requirements, and considerable amounts of money being spent unnecessarily on running stream processing (TUDORAN et al., 2016).

Furthermore, this work proposes and analyzes a series of solutions, which seek to mitigate the problems found in these applications and environments. Specific objectives and solutions are defined to solve or ameliorate these limitations. These objectives are described in the following section, as they are presented separately with their respective approaches.

1.3 Goals

In this section are defined the objectives which are expected to achieve during the conception of this research.

The main goals of this research are described as follows:

- **Provide a generic model of a dispatcher for big data applications, for stream and batch processing:**
  The recent growth and popularity of Big Data have led to the definition of wide range environments and applications, for different contexts and fields. Therefore, it is necessary to establish a generic solution that fits most of the existent scenarios.
- **Employ techniques that optimize the network communication and increase the throughput of messages:**
  Several applications require really tight deadlines (MILOSLAVSKAYA; TOLSTOY, 2016) in order to increase the information value. It is necessary to reduce the latency between the exchange of messages, so different algorithms are presented to achieve this goal.
- **Dispose of efficient approaches for data distribution:**
  Robust environments are composed by several machines that run separately to found a result, so as it is possible to reduce the communication and less overhead will be generated in a environment. Distinct data partition algorithms are presented seeking to reduce shuffling of messages and dependency of data in order to achieve this objective.
- **Increase the properly usage of the available resources:**
  Most of the available infrastructure are heterogeneous and require proper definition of a task scheduling, operator distribution and data placement. Thus, assuming that the available resources have different capabilities i.e. heterogeneous environments, whether CPU, memory, disk and network throughput, it is necessary to select the right approaches to reach properly usage of resources.

The next section presents the methodology which was conduct in this research in order to achieve the objectives already discussed in this section.

1.4 Methodology

During the realization of this research, methods were adopted and criteria were established in order to provide pertinent contributions and quality of content. They are presented and discussed in this section as follows.

The study of the state of the art was conducted by filtering articles, thesis and white-papers by keywords and from contemporary dates, at least five years except for opportune content that is somewhat ten years old. Of course, there is also classic computer science solutions older than ten years old, which were discussed because of its impact in some scenarios. Furthermore, renowned events, conferences, and journals with high impact in the technology community were firstly selected.

After the study of the collected contents, the challenges, the models, and the solutions were classified and compared to propose a complementary model which will contribute to the actual state of the art. As the model is presented, a method is necessary to evaluate this model, whereas a prototype is conceived for the validation of the model. Besides that, every technology and solution is discussed and evaluated in the following chapters.

The performance analysis of the prototype is conducted following the methodology of (JAIN, 1991) to evaluate computer model performance using experimental designs. In order to evidence the results, metrics from state of the art were used to demonstrate the effectiveness and efficientness of the model. Thus, the metric results obtained from those designs were evaluated and validated using statistics. All the results, analysis, and graphics are presented and discussed.

1.5 Organization

The current chapter presented up to a brief introduction of this research pointing motivation, problems, contribution, methodology and its goals, the rest of this document is organized as follows:

Chapter 2 introduces the background of Big Data, discussing basic concepts and the history of Big Data, whereas MapReduce, the processing models and the architectures of Big Data are presented. Chapter 3 introduces the Stream Processing engines, the mechanisms which provide the capacity to compute large amounts of data. Also, displays the Message Queues which are essential components of stream processing systems. Besides, discusses every particularity of these frameworks and solutions that are useful for this proposal.

The related work of this research is presented and discussed in Chapter 4. The related work is separated into three sections: the dispatcher modules, the proposals that use communication optimization and the techniques and proposals of stream partitioning. After this chapter, Chapter 5 presents Aten as the solution model of this research. It is to provide an overview of the proposal, alongside the dispatcher model and the prototype developed during this research.

Thus, the evaluation of this proposal is presented in Chapter 6 providing the details of the methodology and environment where the proposal is evaluated. Moreover, the results are presented and discussed. Finally, the final considerations of the proposal are shown in Chapter 7 alongside the future works.

2 BIG DATA BACKGROUND

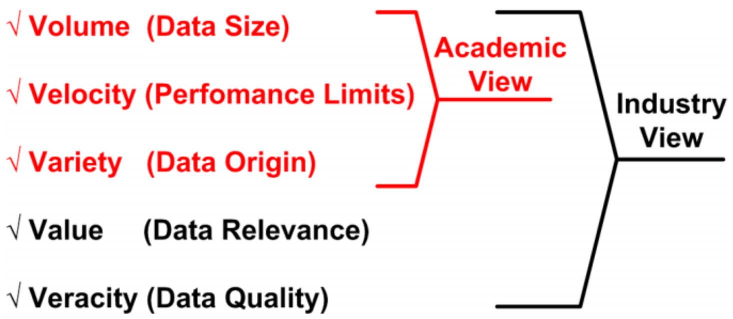
This chapter describes and explains the concept o Big Data from the very beginning of its conception, applicability, the models and the architectures as a complementary introduction necessary to achieve the goals of this research. The first section presents up the overview related to Big Data, as provide examples and brings pertinent numbers of the production of data. The next following sections present, respectively, the MapReduce programming model, the processing modules which implements MapReduce, the architectures and finally a overall discussion about the chapter.

2.1 Big Data Overview

Big Data applications are widely adopted in various social and economic contexts. For instance, Facebook estimates to receive 4.75 billion shares, 4.5 billion likes, 420 million status updates and 300 million photos every day (NOYES, 2015). In addition, financial market applications can be cited, such as Day Trade - where an investor needs to buy stocks with rising values, and then rapidly and repeatedly sell them at the proper time (GAD, 2017). Using a processing mechanism, which can calculate statistical trends within popular trends in social networks, promotes a better understanding of the data and helps in decision-making. Furthermore, the telecommunication industry uses real-time data processing to improve the flow of their call services (HILBERT, 2016).

In the literature, Big Data is composed by different aspects, characteristics or dimensions known as the 5 Vs of Big Data (ASSUNÇÃO et al., 2015), which is illustrated in Figure 2.1. These five Vs compose the idea behind Big Data, which are the criteria that were defined by two different contexts to characterize and accomplish Big Data.

Figure 2.1: Big Data: The 5 Vs. Source: (MATTEUSSI et al., 2017)



The Figure 2.1 presents the 5 Vs of Big Data, it is separated in two different contexts, which is the academic and the industry point of view. The academic concerns about the investigative disciplines that are inherent in how best to dispose of data, improve resource utilization, transfer data load, improve performance and processing limits, and so on. On the other hand, the industry is typically interested in evaluate the quality of data and its financial outcome, the best strategies for mining and monitoring thereof, the relevance of the results obtained and the costs expended for this purpose. Besides, regardless of organization, would be it academic or industry, the 5 Vs are characterized as follows:

- **Volume**: it is related to the magnitude of data which is in constant expanse and the scale it may reach;
- **Velocity**: it refers to the rate at which data is being produced, and how fast they must be treated to meet the processing and analysis;
- **Variety**: the produced data has different natures, for example, e-commerce sites use structured data as logs of web servers are known as semi-structured data, on the other hand social networks may provide unstructured data such as audio, video, images, and so on;
- **Value**: the data in its raw format may not aggregate value, but when it is processed, analyze or cross used with more information it can become more valuable since it can provide strategic information to the business intelligence.
- **Veracity**: it refers to reliability over the data source, such as measuring customer sentiment in social networks is uncertain in nature, since it implies the use of human judgment. Even so, this data contains valuable information that even imprecise and uncertain can be used as training set or historical data to future directions.

These data sets have become a valuable source of information that arouses the interest of many organizations for the potential it can bring to business. Big data has been employed to describe the vast amount, the incredible growth, the applicability and use, and the availability of structured or unstructured information (CHANDARANA, 2014).

One of the greatest and most recent computational challenge is to store, manipulate and analyze high amounts of data, that require fast answers and low budget projects. Data itself can be easily handled in low scale, but this is not the case. Since there is a huge amount of data being produced daily (scale of peta bytes) (MATTEUSSI, 2016), which is called Big Data. It is used by cooperative systems, web services, eletronic devices and social networks (WHITE, 2009) are required capable infrastructure and proper resources to achieve tangible results.

The richness contained in the information obtained through the analysis and processing of data enhances the strategic decision making, in order to produce very valuable information. For this reason, many organizations are looking for efficient, intelligent, reliable and low-cost solutions to handle large-scale data processing and analysis.

Big Data study field become real since the introduction of MapReduce, formulated by Google. In order to achieve these five Vs, the model attended the necessity of the industry and has enabled large files to be processed, so common problems such as memory overflow, disk limits and high execution times could be suppressed. Improving even the use of resources for this processing, reducing costs and providing alternative solutions for all stakeholders. The next section will present the Map Reduce paradigm and provide further details of its idea.

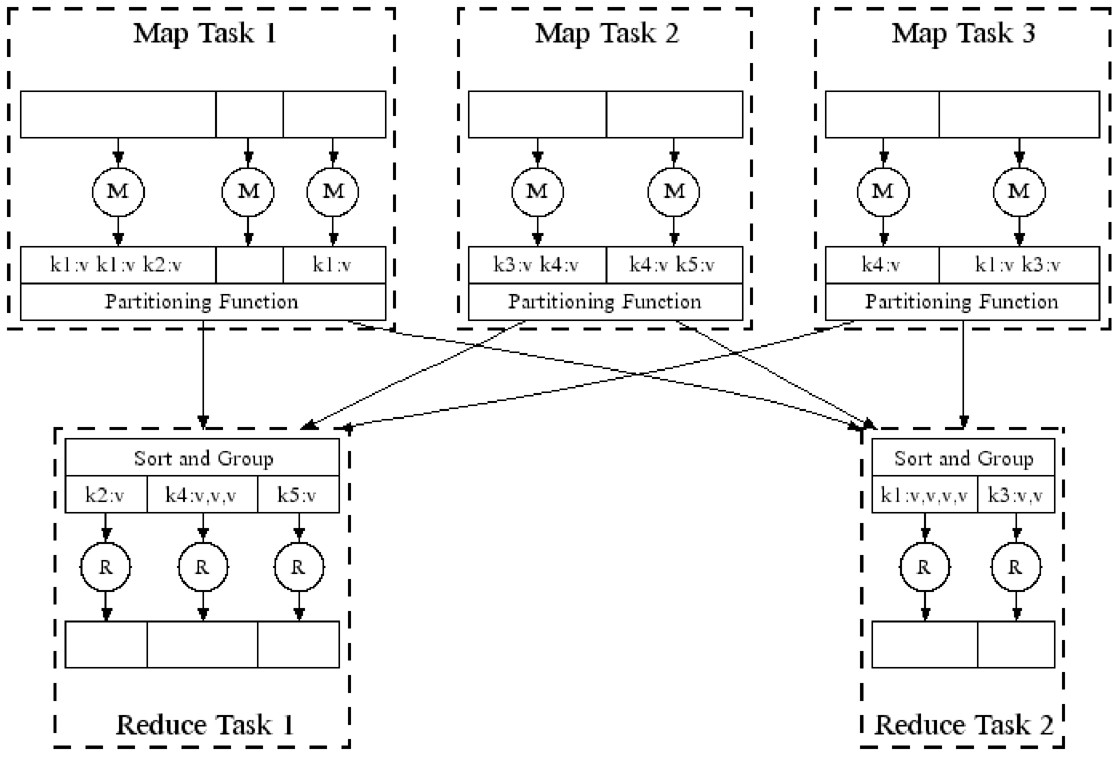
2.2 MapReduce Programming Model

The MapReduce programming model was first introduced in the LISP programming language and later popularized by Google (DEAN; GHEMAWAT, 2008). The model was proposed in 2004 at USENIX Symposium on OSDI (Operating systems Design and Implementation (GOYAL; BHARTI, 2015). The model is quite simple, but troubleshooting to express useful programs. Although it may be a little harsh, the proposal was to abstract some of the tasks of managing parallelism complexity out of the user's responsibility. Alongside that, MapReduce includes synchronization abstraction, fault tolerance, task scheduling and data distribution in a straightforward way.

The MapReduce is based on the *map* and *reduce* primitives, both provided by the programmer. First, *map* function takes a single instance of data as input, which are distributed amongst cluster nodes, and produces a set of intermediate key-value pairs. Second, the intermediate data sets are automatically grouped over or shuffled by keys. Then, the *reduce* function takes as input a single key and a list of all values generated by the *map* function for that key. Finally, this list of values is merged or combined to produce a set of typically smaller output data, also represented as key-value pairs (DEAN; GHEMAWAT, 2008; MATTEUSSI; ROSE, 2015).

Furthermore, there is another step during the MapReduce called *Shuffle*, which exist between the map and the reduce and executes before the reduce phase. The shuffle consumes the output of the maps and guarantees that the input to every reducer is sorted by some criteria (e.g. key), this is necessary to group the common keys before performing

Figure 2.2: MapReduce Parallel Execution. Source: (DEAN; GHEMAWAT, 2009)



the reduce appropriately. If the shuffle is not properly done, will be required a new step of shuffling to rearrange the data, increasing the execution time. The shuffle is the MapReduce's core, an area of the codebase where refinements and improvements are continually being made (WHITE, 2012).

The parallel execution of MapReduce is illustrated in Figure 2.2, during the development or the setup of the execution of a MapReduce application it is possible to define the number of maps and reduces that will run in parallel. In this figure is defined 3 map tasks and 2 reduce tasks, the map prepares the dataset separating in key-value as discussed earlier, then the partitioning function separates the workload to the shuffle phase, which sorts and groups the keys to the reduce phase. Of course, there is more going on in applications that use map-reduce but is a straightforward manner to explain the map reduce flow.

Considering the triviality of the parallelism for map reduce applications, still, there are open challenges such as defining a model to define the correct numbers of map and reduces tasks ideally in order to optimize those applications. Alongside that, map reduce assumes that the workload is balanced and will be split fairly which is not always the case, principally where there is skewed data and heterogeneous environments (e.g., Volunteer

Computing, Grid, Private and Public Cloud) (WHITE, 2012). Besides, it is essential to pinpoint that cloud computing has become a robust platform to perform large-scale and complex computing such as map reduce, necessary to conduct operations in the Big Data level. Cloud scenarios provide advantages such as security, efficiency, flexibility, pay-per-use methods and scalable data storage (HASHEM et al., 2015) (RISTA et al., 2017), but promoting a situation of interference which is not beneficial for map reduce tasks.

2.3 Big Data Processing Models

It is ideal for Big Data processing that resembles large datasets - composed of structured and unstructured data. Big data enables users to process and analyze distributed queries across multiple sites, returning resultant sets promptly through the use of Batch and Stream Processing models. Big Data processing models are separated in two subclasses, the batch and the stream models. Recently, there is a increase in stream popularity due to the requirement of real-time and near real-time responses.

2.3.1 Batch Processing

Batch processing, historically, was one of the first model to use the MapReduce paradigm. It consist of several batches that derives from a large data entry which is divided into smaller pieces called chunks, typically from 32 MB to 64 MB. The chunk size can be specified by a parameter provided by the programmer, although the value depends on the technology of the disks, resource capabilities, data distribution algorithm and the data transfer rate from memory to disk (WHITE, 2009).

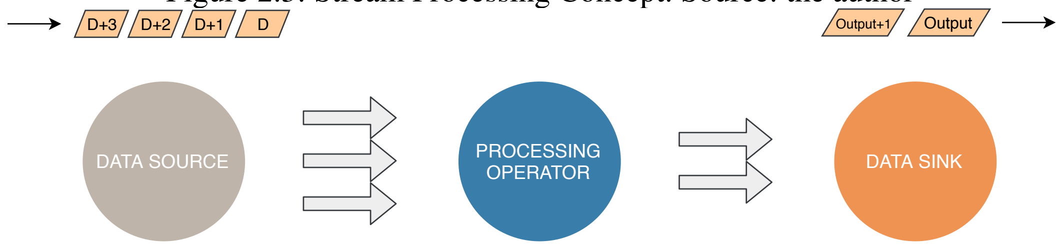
The Apache Hadoop framework was one of the first's implementation of the MapReduce paradigm in order to process batch data(BORTHAKUR, 2007). In Hadoop 1.x, for instance, map reduce tasks are wrapped in CPU cores, processed in waves, and consolidated in the disk when completed. In this case, it is possible to observe Hadoop providing only CPU resource sharing. On the other hand, the YARN, which is a resource manager, matured the Hadoop ecosystem allowing support for a wide range of frameworks, applications and its heterogeneous workloads. YARN decomposed into fine-grained, loosely-coupled parallel tasks, scheduling on task-level rather than on job-level, improving fairness and utilization of resources, but unfortunately allowing only CPU and memory resource management (VAVILAPALLI et al., 2013). Furthermore, Hadoop implementations and batch processing frameworks neglect disk and network I/O management, overloading OS I/O schedulers and generating performance degradation when processing data-intensive applications in large scale.

In order to process big files, batch processing usually depends on file system replication, in the Hadoop Framework is used the HDFS (the Hadoop Distributed File System) which has many similarities with existing distributed file systems. However, the differences from other distributed file systems are significant. HDFS is highly fault-tolerant and is designed to be deployed on low-cost hardware, which is the necessary in order to process large data in batch. Besides, HDFS provides high throughput access to application data and is suitable for applications that have large data sets. HDFS relaxes a few POSIX requirements to enable streaming access to file system data. Those differences are necessary to reproduce the batch processing using map reduce in high parallel and distributed environments.

2.3.2 Stream Processing

Stream Processing model was initially used in the control and automation sectors, in the financial sector - stock exchange and electronic commerce, and is currently already used by many organizations and for the most varied purposes (MATTEUSSI et al., 2017). In SP scenarios, thousands of sources generate data, continuously and simultaneously with unpredictable flows. Usually, this data is relatively small (in kilobytes unit) and has varied types. We can cite some examples such as client-generated behavior from log files, sensors, mobile applications, web applications, e-commerce shopping, player activity during a game, social networking information, and so on (ANJOS et al., 2015; AMAZON, 2016).

Figure 2.3: Stream Processing Concept. Source: the author



The natural perception of a data stream processing paradigm is through the modeling of data flows by means of a graph. The graph contains source of data that continuously emit items which are processed by connected nodes that apply some function, for instance an ETL like operation, in each item (MORIK, 2014). That was the main concept of this programming model, which is commonly splitted in the major elements (MORIK, 2014; BUDDHIKA et al., 2017) presented in the Figure 2.3: the data source (i.e. Ingestion Operator) which collects or produces data; the processing node, which computes each element that are emitted by the data source; and the data sink, which stores, persists or sends the result of a data stream.

The graph in streaming applications are usually characterized by DAGs, where the vertices are operators and edges are streams and has no directed cycles. In addition, some stream graphs might be cyclic, depending on the application requirements (HIRZEL et al., 2014). The operators can be used for stream ingestion, computation, or data sink (BUDDHIKA et al., 2017).

1. **Ingestion**: the operators tend to be stateless, being easier to scale and replicate;
2. **Computation**: most of operators are stateful (Map, filters, ...) and the outcome depends on the built up state, implying difficultness to scale and parallelize, and resulting in possible bottlenecks;
3. **Sinks** Data sinks are the final destination in a stream flow, normally responsible for redimensioning and storing data for further analysis and visualization.

Essentially, as ingestion operators do not represent bottlenecks, a previously discussed constraint is the transfer between points, which is the end to end data transfers that happen before the ingestion operators. It principally suffers from network interference and congestion, resulting, for instance, in high latency. At this point, it provides an opportunity to overcome the potential of conceiving and acquiring events through accumulation and transfer within data blocks (i.e., considering the time constraints given by the application), allowing to reduce the latency transfer cost and enabling a higher throughput of events over time.

Furthermore, SP processing occurs continuously, sequentially or incrementally within events or by window times (a time slice). It produces a broad range of analytically data as correlations, aggregations, filters, and sampling (AMAZON, 2016). Inside the category of Stream Processing can be found two different architectures, which is the Lambda Architecture and the Kappa architecture. Both of the architecture are shown in the following section.

2.4 Big Data Architectures Overview

Big Data applications face a dilemma; it is required powerful resources to attend the requirements of velocity and value, since to provide value to the outputs is necessary to do so fastly. The standard solution is to increase the number of available resources, so the cloud computing has become a robust architecture to perform large-scale and complex computing. Although the costs rise, because of solutions that are not proportionally scalable.

Cloud scenarios provide advantages such as security, efficiency, flexibility, pay-per-use billing structure and scalable data storage (HASHEM et al., 2015) (RISTA et al., 2017). For instance, in order to address security in Big Data processing in the cloud, (ANJOS et al., 2017) proposed an infrastructure to allow many users with different permissions to share a cloud environment safely. Also, the cloud decreases infrastructure maintenance expenditures, as well increases in an efficient way the management, and improves multi-tier user access. As a result, the number of applications hosted in cloud platforms has been growing faster over the past years (HASHEM et al., 2015).

Due to the cloud adoption, data generation rates increased, leading to the Big Data age. It resembles large datasets, composed of structured and unstructured data that need real-time analysis. Moreover, Big data enables users to process distributed queries across multiple datasets and return resultant sets promptly. Finally, cloud computing provides the underlying engine through the use of Stream Processing models. Of course, there is still space to enhance these models in order to provide better exploitation of cloud resources. The follow subsections present the architectures used in public and private cloud in order to process Big Data.

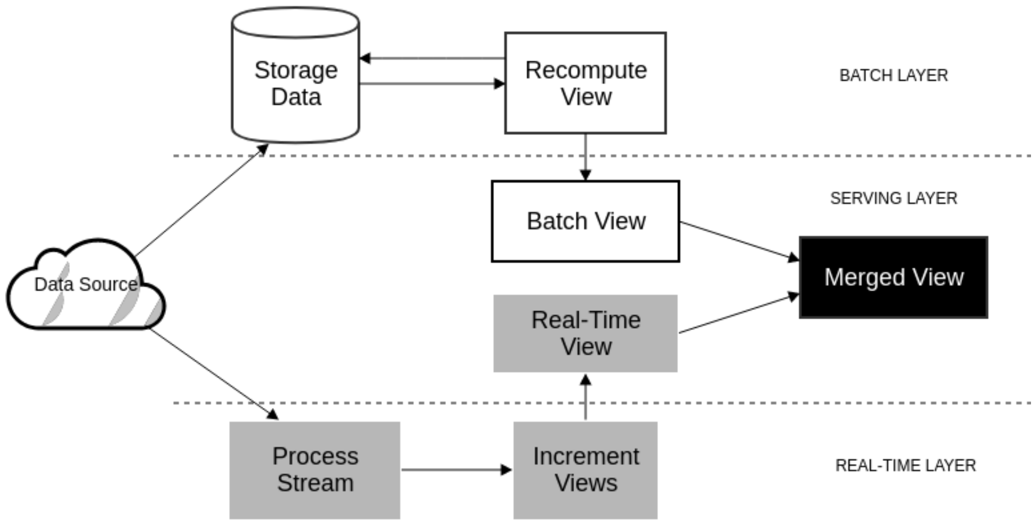
2.4.1 Lambda Architecture

Marz proposes Lambda as an architecture for batch and RT processing. This architecture attempts to balance latency, throughput, and fault tolerance while using batch and RT processing to provide comprehensive, accurate and real-time view of the data (MARZ; WARREN, 2015).

The motivation behind this architecture is the need for a robust system that is fault-tolerant (for hardware or the human factor), serve a solution for a wide range of workloads and use cases, to be linearly scalable, and to become the system extensible so that features are properly inserted as to reduce maintenance (HASANI; KON-POPOVSKA; VELINOV, 2014).

The lambda architecture is not just a architecture, it is also presented as a software design pattern, which unifies online and batch processing within a single framework. It enables users to optimize their costs of data processing, by understanding which parts of the data need online or historical processing, and by self-adapting during the collect and analyze of data (KIRAN et al., 2015).

Figure 2.4: Lambda Architecture. Source: Rewritten by the author (MARZ; WARREN, 2015)



The Lambda architecture can be seen in Figure 2.4. The architecture is composed by three main layers that are presented as follows:

1. The batch layer stores the master copy of the dataset and precomputes batch views on that master dataset. It manages large (immutable) data sets, usually historical data, which are is constant growth and are computing folowing arbitrary functions. Pre-processed in arbitrary queries (views), typically done by frameworks as the Hadoop (WHITE, 2009), Hive (HIVE, 2015), Pig (PIG, 2015) and Spark (ZAHARIA et al., 2016). Besides, Marz indicates in his book that the batch layer is simple to use, since batch computations are single-thread and trivial to obtain high level parallelism. In other words, it is easy to write powerful algorithms that are highly scalable.
2. The RT layer processes small data sets according to a time window (for example, 1 minute). This strategy allows real-time processing using incremental algorithms. This layer updates whenever the batch layer finishes preprocessing a view, this is necessary to ensure that new data is represented in query functions as quickly as necessary, either for meeting the application's requirements or to provide consistency

on its results. The batch layer is quite similar to this layer, both produces views based on data it receives but the speed layer only looks at recent data, whereas the batch layer processes everything (MARZ; WARREN, 2015). This is necessary since the speed layer attempts to provide low latency during the execution of queries.

3. The service layer combines the results of the batch and real-time processing layer to allow interactive (non-latency) user analysis. This layer can use relational databases, but also non-relational databases. This layer is a specialized distributed database that loads in a batch view and makes it possible to do random reads on it. Whenever there is new results available, the layer automatically swaps the views. Since this layer supports batch updates and random reads and do not support random writes this databases are extremely simple, it enables the robustness, simplicity and easy configuration to operate in harmony (MARZ; WARREN, 2015).

Marz indicates that the beauty behind the Lambda Architecture is that once the data go through the batch layer to the serving layer, the results from the RT views are no longer necessary. It means that the real-time view parts can be discarded, which is ideal, since the speed layer is far more complex than the rest of the layers. Marz classifies it in the architecture as the *complexity isolation*, whereas the complexity is pushed into the layer whose results are only temporary. If something ever went wrong, those states and results could be discard and everything will be available to normal within a few time.

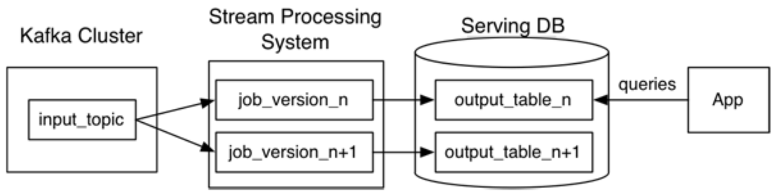
2.4.2 Kappa Architecture

The Kappa Architecture was proposed in 2014 by Jay Kreps, one of the principal creators of Apache Kafka (LIN, 2017), a messaging and brokering system. In contrast to Lambda Architecture, the Kappa architecture is very similar but works only on service and real-time layers. Kappa simplifies Lambda architecture avoiding data replication and providing event-processing (devices' data streams), whereas everything is a stream and therefore requires only a stream processing engine.

The author of the Kappa architecture says that although Lambda Architecture solves important matters that are generally ignored, he does not believe that Lambda is a new paradigm or will be the future of Big Data, considers that is just a temporary state caused by the limitation of tools (KAPPA. . . , 2014).

The figure 2.5 presents up the Kappa Architecture. It is a single layer to process

Figure 2.5: Kappa Architecture. Source: (KAPPA. . . , 2014)



RT information. The Kafka Cluster indicated in the figure is responsible for retaining the full log of the data, which permits to reprocess and contains multiples subscribers which enable several consumers. Thus, whenever the reprocessing is necessary, Kappa suggest starting a second instance of your stream processing job that begins processing from the very beginning of the retained data, inserting in a new output location. Once the data is processed, the appointments switch the application to read from the new location (KAPPA. . . , 2014). This approach requires the double size of the store needed in Lambda since it consists in maintain part of the information.

The idea behind the proposal of Kreps was to provide a simple architecture to process RT events, which is indicated in situations where is necessary use a single framework. There is no real difference of efficiency or any advantage comparing to Lambda, but to provide an alternative and simple solution to people to operate with an different perspective, principally in scenarios where simplicity matters. It important to emphasize that Kappa is also a popular alike architecture of setups for Big Data solutions (ANJOS et al., 2015; ASSUNÇÃO et al., 2015).

2.5 Discussion

The recent growth and popularity of Big Data have led to several challenges, since the solutions that were required to evolve, following the constant demands. For instance, the prevalence of stream processing and the requirements of real-time bought the principals problems in batch, like the high latency to process the vast amounts of data and therefore a conception of the new model of stream processing. Whilst, the business point which requires the now famous Business Intelligence that bought requirements of ETL and complex structured data relations in the Big Data scope. These lead to the necessity of a hybrid processing engine which crosses the data stream with the batch in order to process historical data or to apply the popular Machine Learning algorithms to provide insights about the now and the future.

Alongside that, problems such as the lousy exploitation of resources have to lead to additional spare of money, creating a new opening to the research in that area. The next chapter presents the Stream Processing Engines, this study focuses mainly on the stream engines that can be integrated with batch processing, since they still lack in unique solutions and enchantments to achieve better results and suppress the community restrictions.

3 STREAM PROCESSING ENGINES

In this chapter are discussed the perspective of concepts and infrastructure for Stream Processing Engines implementations in homogeneous and heterogeneous environments. The introduction and presentation of characteristics of each framework engines that are proposed as means of reaching the Big Data Analytic required in recent stream applications. Alongside that, the strategies adopted for different Big Data Frameworks are summarized and discussed regarding the problems that this research intends to overcome. Besides that, another vital part of the architecture of SPEs is also presented and discussed: the Message Queues.

3.1 Overview

The very idea of providing insight in real time is not entirely new. The SPS was already present in the academia and the industry for quite a while and are categorized in three generations (JAVED; LU; PANDA, 2017): the first generation which are the databases systems and rule engines using procedures and triggers(STONEBRAKER; ROWE, 1986); the second generation consists of operators that are SQL based, but limited to a single node (AKIDAU et al., 2015; ABADI et al., 2005; BARGA; CAITUIRO-MONGE, 2006); and finally the actual generation of big data engines that process data (i.e., frameworks), which process the data streams using different approaches and techniques with higher scale (STORM, 2014; Apache Spark, 2014; KATSIFODIMOS; SCHELTER, 2016).

The Stream Processing Engines consumes data stream, which is a continuous pipeline of unbounded data, the termination point of which is not predefined. It usually consists of messages that are separated into key-value pairs of records (JAVED; LU; PANDA, 2017). The data stream incoming flow is not typically continuous, and it variates over different scenarios or applications (XHAFA; NARANJO; CABALLé, 2015), which is related on the production of data from multiples sources such as IoT, Social Media and so on.

Furthermore, there are several concerns related to Big Data frameworks since they can have a high variability of flaws (e.g. infrastructure failures, unavailability of services, resource overflow, network instability). For instance, it is a concern in stream processing when a crash happens in an operator, which may cause loss of data. Those operators can be stateless or not, and when it holds a state, this state is molded by the incoming flow of a data stream, and if it crashes during the consumption of a data stream, those will be lost (JAVED; LU; PANDA, 2017).

3.2 Big Data Platforms

Frameworks for stream applications typically require environments that are capable of supporting large-scale data processing, structured and unstructured data, high data ingestion rates, and resource availability variations. Under this circumstances, cloud computing enables resource sharing, scalability, and resiliency to address these challenges. Nonetheless, recent works (ABAWAJY, 2015) met the following problems: network contention (e.g., high latency), resources limitations (i.e., CPU, Memory, Disk, Virtual Machines per datacenter), and high costs to process Big Data Streaming applications on clouds environments.

The main concern of a Stream processing frameworks is to achieve two important attributes, which is low latency and high availability of the system (JUNIOR et al., 2018). A batching processing framework usually lack on those attributes, therefore new frameworks were proposed. These new frameworks usually attempt to achieve those attributes providing different solutions that are presented in this section. Not only that, but real application often require to cross process historical data i.e. big batch files, consequently this research seeks to mitigate proposals that converge in that scenario, in order to propose different solutions to achieve its goals.

3.2.1 Hadoop Streaming

Hadoop (HADOOP, 2009) is an open-source version of the MapReduce model and the HDFS, a distributed file system that provides resilient, high-throughput access to application data (BORTHAKUR, 2007). Besides, we can cite the Hadoop Streaming that enables real-time processing and supports the programming languages Python, PHP and Ruby among others (MATTEUSSI, 2016).

Currently, there are two different versions releases of Hadoop: 1.x and 2.x. The first is the original Hadoop implementation. The second is intended to be the next generation of Hadoop, called MapReduce 2.0 (MRv2) and is coupled with YARN (Yet Another Resource Negotiator), which is considered a resource management system that enables multiple frameworks to run on the same physical cluster. Later on, a stream version of Hadoop was released. It is called Hadoop Streaming and it has provided several features to implement near real-time and real time analysis using transformations, operators, data source streams, partitioning and so on.

Programming in Hadoop Streaming is still a little more complex than the original MapReduce programming norms, because the programmer has to learn the whole Hadoop APIs and Java programming language, handle specific types (classes) of (Key, Value) pairs and know how to initialize a MapReduce job (DING et al., 2011). Map function consumes a pair of (Key, Value) and produces zero or more intermediate (Key, Value) pairs:

$$Map(k1, v1) \rightarrow list(k2, v2) \tag{3.1}$$

Reduce function takes a pair groups of (Key, Value), merges the values in the list, and outputs possibly a smaller set of values:

$$Reduce(k2, list(v2)) \leftarrow list(v3) \tag{3.2}$$

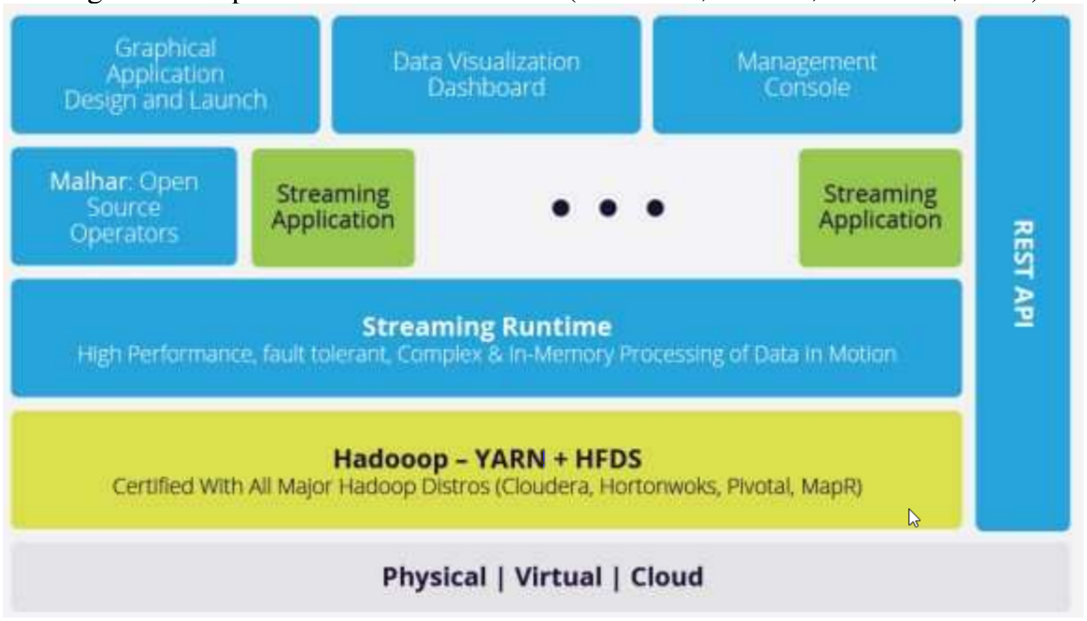
Furthermore, in the API of Hadoop Streaming there is no alternative to enable the management of stream partitions, it is possible to manage jobs but in its core, the whole engine is moved by the YARN implementation. So, there is no guarantee that the management is well-done, but the believe that YARN is well-prepared to conduct the tasks on its finest.

3.2.2 Apache Apex

Apache Apex is presented as a powerful, reliable, versatile and flexible stream processing hybrid platform for Big Data. Apache Apex provides a SPE that has in-memory performance and scalability, fault tolerance and state management, native rolling and tumbling windows support, hadoop-native YARN and HDFS implementation (PATHAK; RATHI; PAREKH, 2016). The Apex architecture is presented in Figure 3.1, which is an adaption from its article (PATHAK; RATHI; PAREKH, 2016). The architecture is composed by six principal components that are described as follows:

1. The input data manager, that expects input from Physical, Virtual and Cloud systems.
2. The Hadoop core: comprised of YARN and HDFS forming the basis of Streaming

Figure 3.1: Apex Architecture. Source: (PATHAK; RATHI; PAREKH, 2016)


Graphical Application Design and Launch
Data Visualization Dashboard
Management Console
Malhar: Open Source Operators
Streaming Application
Streaming Application
Streaming Runtime
High Performance, fault tolerant, Complex & In-Memory Processing of Data in Motion
Hadooop – YARN + HFDS
Certified With All Major Hadoop Distros (Cloudera, Hortonwoks, Pivotal, MapR)
REST API
Physical | Virtual | Cloud

Applications.

3. Streaming Runtime: In-memory processing of data in motion alowing the usage of time windows.
4. The Malhar[1]: Library of open source operators developed to use in Apache Apex robust platform.
5. Streaming Application integration that allows the business logic through DAG.
6. User interface and API: allowing dashboards and management of application

The general proposal of Apex is to provide an framework that integrates the benefits of Hadoop allowing to process Stream and Batch simultaneously, with simple code integration and easy to use. This is possible using the Malhar Library with possibility of total customization, granting total independence to developer to manage operators and infrastructure. Furthermore, as part of its proposal it is possible to develop for Apex using JavaScript, Python, R, Ruby and Java.

Notwithstanding, the Apache Apex depends on the Hadoop implementation, either from the management of YARN but also from the distribution of files in HDFS. There is no further details related to the management of stream partition or data partition on its implementation.

[1]https://apex.apache.org/docs/malhar/

3.2.3 Apache Storm

The Apache Storm (STORM, 2014) enable real-time analytics, online machine learning, distributed RPC (remote procedure call) and more. Traditional MR applications are expected to start and finish, but this behavior differs slightly in Storm. Storm continuously processes messages at real-time (XAVIER et al., 2016).

In this framework the applications are called topologies and are built in the form of DAGs, with *spouts* acting as data sources and *bolts* as operators. The communication between them occurs through *streams*. Each component can declare own *streams* to be produced and their schema, but only *bolts* can consume the *streams*. Stream grouping provides a parallelism abstraction level, since all the operators are stateless it is possible to run multiple tasks simultaneously (avoiding bottlenecks) (LI et al., 2016).

Moreover, Storm guarantees messages processing completely. It is made by *spouts* that places messages in a queue until there is a confirmation that they have been processed, and if a message fails the *spout* sends it again (GRADVOHL et al., 2014). The abstraction of Spouts and Bolts allow to manage or manipulate the stream data, but it does not provide clear vision from the resource that are being used.

Apache Storm master node aka nimbus, is a process that runs on a single machine under supervision. In most cases the nimbus failure is transient and it is restarted by the supervisor. However sometimes when disks fail and networks partitions occur, nimbus goes down. Under these circumstances the topologies run normally but no new topologies can be submitted, no existing topologies can be killed/deactivated/activated and if a supervisor node fails then the reassignments are not performed resulting in performance degradation or topology failures. Furthermore, nimbus is coordenated by the Zookeper manager. Besides, the workers ran inside multiples JVMs and are managed by a supervisor which talks directly with a Zookeper manager, and handles distribution and synchronization of tasks in order to avoid failures.

3.2.4 Apache Spark

Apache Spark was introduced in 2012 as a new cluster computing framework that can run applications up to 40x faster than Hadoop, by only keeping data in memory. It can use interactively queries in larges datasets with sub-second latency (ZAHARIA et al., 2012).

Spark (Apache Spark, 2014) is considered a powerful and one of the most used framework for real-time processing and data analysis. It was initially developed to support in-memory process applications for a varied range of solutions such as interactive data mining and algorithms, graphs, machine learning and so on (ZAHARIA et al., 2013). Spark uses memory-based abstraction called RDDs which allows to process data using small batches by time intervals (windows) to provide high performance.

RDD provides an interface based on coarse-grained transformations that carry out the same operation for many data items, which are collections of objects partitioned across cluster nodes. This allows them to efficiently provide fault-tolerance by logging the transformations used to build a dataset rather than the actual data. If a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to recompute just that partition. Thus, lost data can be recovered, often quite quickly, without requiring a costly replication.

The general execution model of Spark can optimize arbitrary operator graphs, and it supports in-memory computing, which lets it query data faster than disk-based frameworks like MapReduce (HADOOP, 2009). Finally, it is offered some APIs for Java, Scala, Python and also can be performed in YARN, Mesos, *standalone* or in the cloud.

Figure 3.2: Spark Architecture. Source: (INC, 2016)

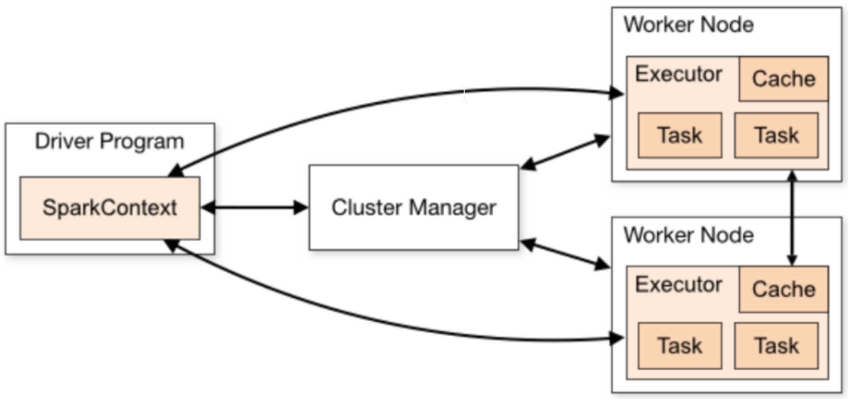


Figure 3.2 shows the architecture behind the Apache Spark framework, it is separated in three components: Driver Program, Cluster Manager and the Worker Node. Inside the Driver Program runs a spark context, which is global scope of any application for the principal program or driver, it communicates with the cluster manager and the working nodes to coordinate each instance process. The Cluster Manager manages the available resource to run Spark, it may consist from different resource managers e.g. YARN or Mesos (Apache Spark, 2014). And finally the worker nodes, which have instances of executors that perform a operation or many operations, side by side with a shared memory

cache.

It is also possible to use spark for batch processing on large datasets, including ETL workloads to convert data from a raw format (such as log files) to a more structured format and offline training of machine learning model (ZAHARIA et al., 2016). It is all feasible by the RDDs that build on MapReduce ability to emulate any distributed computation but make this emulation significantly more efficient. Although, the main limitation of the RDD is the latency, always a problem in stream applications, due to synchronization during the communication. Besides the network and disk bandwidth, another limitation is the available memory, that usually is all consumed or bottlenecked by the CPU.

3.2.5 Apache Flink

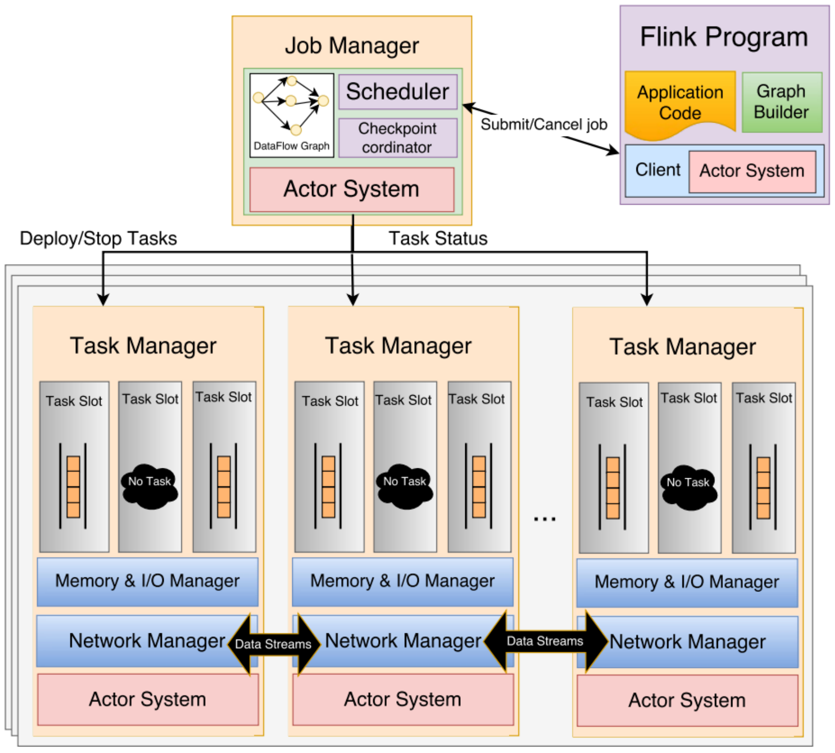
Apache Flink (CARBONE et al., 2015) follows a paradigm that embraces data-stream processing as the unifying model for RT analysis and batch processing. RT processing uses a highly flexible windowing mechanism, Flink programs can compute early and approximate, as well as delayed and accurate, results in the same operation, obviating the need to combine different systems. The fault tolerance module consist of state system checkpoints, which is inspired by the Chandy-Lamport algorithm (CHANDY; LAMPORT, 1985).

Figure 3.3 describes the components of Flink and their interactions. Every time a flink program is submitted the Job manager (i.e. the master) decomposes the job in smaller tasks and schedules them on the available task managers. The Task Manager are the slaves and are responsible to process and interact to each other in order to achieve a result, which later on is returned to the job manager(JAVED; LU; PANDA, 2017).

At the same time, Flink needs dedicated batch processing (dealing with static data sets). It occurs since complex queries over static data are still a good match for a batch processing abstraction (CARBONE et al., 2015). Finally, Flink also has a high-performance execution module and provides automatic code optimization and native support for incremental or non-incremental iterations (ALEXANDROV et al., 2014).

Furthermore, the Apache Flink is indicated as an ideal framework for performance optimization in the presence of data skew (KATSIFODIMOS; SCHELTER, 2016). This framework allows to manage physical stream partitions, it provides methods of repartition and allow to implement different forms of data stream partition. The commons partition methods in the framework are *Rebalance* and *Rescale* partitioning.

Figure 3.3: Flink Architecture. Source: (JAVED; LU; PANDA, 2017)



Besides, the Apache Flink supports iterations in the data pipeline, allowing that the data produced in a point to be fed back to another pipeline for subsequent iterations (JAVED; LU; PANDA, 2017). Therefore, Apache Flink is highly suitable for iterative and machine learning applications.

3.2.6 Apache Samza

The Apache Samza is a stream processing framework that was built on top of Apache Kafka. It uses the Apache Kafka for the management of messages and brokering services, and YARN to provide fault tolerance, resource isolation, security and resource management (KLEPPMANN; KREPS, 2015).

Different from many other stream processing frameworks, Apache Samza does not implement its own network protocol for transporting messages from one operator to another. Instead, a job usually uses Kafka topics to manage inputs and outputs (KLEPPMANN; KREPS, 2015). Also, the partition distribution is managed by kafka, whereas is possible to physically alternate the partitions in order to achieve better load distribution.

The Samza high level API provides a unified way to handle both streaming and batch data. You can describe the end-to-end application logic in a single program with operators like map, filter, window, and join to accomplish what previously required multiple jobs.

The fault tolerance present in Apache Samza is provided by YARN and by RocksDB[2]. It uses embedded key-value store and has low latency and high throughput in data access. Besides, in order to consolidate the state of operators in the face of disk or node failures, every write to the store is also replicated in a topic in Kafka.

3.3 Message Queues

Message queues in the architecture perform an important role, since all the data is available and maintained in those queues. Therefore, the data from this structure must be stored and replicated. This section discusses the messaging broker frameworks application, its role in Big Data architectures and the methods of communication behind these engines.

The architecture of a SPE is complex, then a substantial control over the communication must be structured. The main purpose is that it must control the overhead and make the communication efficiently over the network growing, giving elasticity. Communication is not the only focus of this research, but it is contemplated due its importance in the development of the Dispatcher.

RabbitMQ is written in Erlang and it is built on the Open Telecom platform, and it is based on the Advanced Message Queuing Protocol (AMQP). RabbitMQ is used as a broker to control the sending and receiving of messages. It can be deployed as a queue-like system, or as a publisher-subscribe message system using an exchanger. It is possible by creating a group of publishers and subscribers that can access the messaging nodes, it can create a network of information that can span from small to large scale in a local area, or over large geographical distance.

The distribution of information sent from the publishers to the hub to be distributed to the necessary subscribers allows for applications to run while relying on data from other locations. This allows RabbitMQ to be useful in designing architectures for small localized systems to large, geographically dispersed interactive systems (JONES et al., 2011).

RabbitMQ abstracts the distribution of groups in a local network, forming one logical broker. It allows the implementation of features such as load balancing and fault toleration (IONESCU, 2015). By default, the messaging system distributes messages following a Round Robin algorithm, and after consume this message is deleted from the

[2]http://rocksdb.org/

queues. Thus, through the use o the AMQP protocol is possible to accept connections between different platforms (e.g., MSMQ (Microsoft Messaging Queue) and STOMP protocol).

ActiveMQ is a open source message-oriented middleware (MOM) from the Apache Software Foundation that provides high availability, performance, scalability, reliability, and security for enterprise messaging (SNYDER; BOSANAC; DAVIES, 2017). The ActiveMQ implemented over the JMS spec, written in Java, it provides broker clustering which allows the creation of several brokers that work as a federated network for scalability purposes.

ActiveMQ has loose coupling for application architecture, which is commonly introduced as a way to mitigate the classic tight coupling of RPC. The loosely coupled design is asynchronous and no interdependence or timing requirements. Besides, the applications can rely upon ActiveMQ's ability to guarantee message delivery although it may reduce message exchange performance. The fault tolerance may be provided by Apache Zookeper, but the main framework does not provide replication.

Apache Flume is a stream messaging system that was first introduced by Cloudera's CDH3 in 2011. It consist of a federation of worker daemons configured from a centralized master orchestrated by the Zookeper[3] (a coordination system) (D'SOUZA; HOFFMAN, 2013).

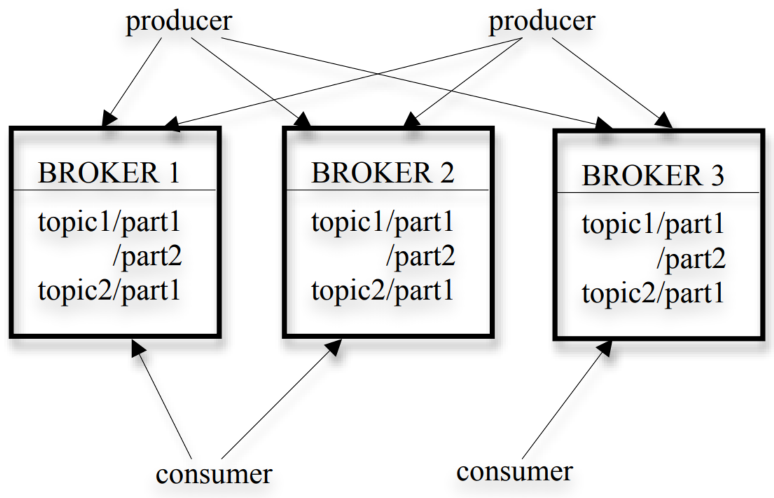
The Apache Flume is composed by channels, which are a construct used between sources and sinks. It provides buffers in-memory for in-flight events, before they be consumed by the stream processing engines. There is also the possibility of using a hybrid configuration of Apache Flume, which allows the store of the in-flight messages in disk, but without the acknowledging of receipt of the event by the sender.

Apache Flume was initially designed to send data to HDFS and HBase. Apache Flume has specific optimizations for HDFS and integrates with the security of Hadoop. Moreover, it is important to note the lack of fault tolerance in Apache Flume, if an agent fails in Apache Flume, the events will be lost until it can recover the disks (KREPS; NARKHEDE; RAO, 2011).

Apache Kafka is distributed and scalable message system, that offers high throughput (KREPS, 2011). Kafka provides a distributed, partitioned, replicated commit log service available by an API that allows applications to events in real time. The Apache Kafka has been open sourced and used successfully in production at LinkedIn for more than 6 months (THEIN, 2014).

[3]Zookeper is a service for maintaining configuration information, providing distributed synchronization. Available at: https://zookeeper.apache.org/

Figure 3.4: Kafka Architecture. Source: (KREPS, 2011)



In Kafka, a stream of messages for a queue of a specific data flow is defined by a topic. A producer can publish messages to a topic. The published messages are then stored at a set of servers called brokers. A consumer can subscribe to one or more topics from the brokers, and consume the subscribed messages by pulling data from the brokers (KREPS, 2011).

The Kafka architecture is presented in Figure 3.4. Assuming that we can distribute Kafka, the Kafka cluster consists of multiple brokers. In order to balance load, a topic is divided into multiple partitions and each broker stores one or more of those partitions (THEIN, 2014). Zookeeper is used as a manager in Kafka for the following tasks: (1) managing the groups of brokers and consumers, (2) triggering a re-balance process in each consumer when the above events happen, and (3) maintaining the consumption relationship and keeping track of the consumed offset of each partition.

Furthermore, Kafka's API provides functionalities that are important in stream processing systems, such as efficient transfer approaches allowing to retrieve multiples messages upon a certain size, usually in kilobytes, avoid explicitly caching messages in memory and very little overhead in garbage collecting its memory, making efficient implementation in a VM-based language feasible.

Mosquitto is a message broker that implements the MQTT protocol, which provides standards compliant server and client implementations of the MQTT messaging protocol. MQTT uses a publish/subscribe model, has low network overhead and can be implemented on low power devices such micro-controllers that might be used in remote Internet of Things sensors (LIGHT, 2017). As such, Mosquitto is intended for use in all situations where there is a need for lightweight messaging, particularly on constrained devices with limited resources.

The architecture pattern client-server present in Mosquitto has many assets, e.g. highly scalable solution. the message header is short in MQTT and smallest packet size in 2 bytes, making it ideal for small devices (KODALI; SORATKAL, 2016). The MQTT offers different forms of message delivery: 1) at most once, 2) at least once and 3) exactly once.

However, Mosquitto does not provide robustness that is presented in Apache Kafka, since the broker system is centralized and the consumers compete during the consumption, although provides guarantees necessaries in IoT scenarios.

3.4 Discussion

For sure there are plenty of frameworks available to process robust amounts of files, each one has its perks and advantages in order to set up different applications. In this section are presented side-by-side some of those characteristics, comparing the benefits for each Big Data framework. There are several attributes in Big Data frameworks which often interest the industry or the academy. The characteristics to benchmark these frameworks were based not only in the point of focus of this research but also to inform and bring pertinent information for future works, for architectures or for models that required different requirements. Alongside these Big Data frameworks, the messages queues are presented as part of their architecture, therefore this section also discusses the message queues available and their convenient perks to reduce constraints in the Big Data scenario.

Table 3.1 shows some features of message handler frameworks. In this table are attempted to pinpoint the main differences between the tools, and with it identify the best engine for hybrid processing model with heterogeneous infrastructure. The main implementation core of each framework is Java, with exception of Spark and partially Storm. Usually every solution that scales and provides sufficient benefits to process large amounts of data is implemented within the JVM(STORM, 2014).

Currently, many large companies adopt these technologies, especially those that have real-time or near real-time requirements. It is noticeable that the popularity of each framework is defined by the size of the companies that adopt its use, this use is usually linked to the ability of the framework to meet the needs of these companies and to have the necessary optimizations to ensure the quality of service.

Besides, in Table 3.1 are shown the languages that are supported by these engines,

Table 3.1: Comparative Between Big Data Frameworks

	Spark	**Storm**	**Flink**	**Samza**	**Apex**	**Hadoop**
Implementation	Scala	Java/Clojure	Java	Java	Java	Java
Companies	Amazon, Baidu, eBay, NASA, Yahoo	Alibaba, Mercado Livre, Spotify, Twitter	LinkedIn, Telefonica, Uber	Netflix, Trip Advisor	SoundCloud, DataTorrent	Amazon, Adobe, eBay, Facebook
Languages	Java, Scala, Python, R	Java, Ruby, Python, Perl, Javascript, PHP	Java, Scala, Python	Java	Java, Python	Java, Python, SQL
Libraries	Machine Learning, SQL, Graphs, Streaming	io-storm (Perl), RedStorm (JRuby), Python, Ruby, storm-node (Node.js), Storm-PHP (PHP), ScalaStorm (Scala), SQLstream (SQL)	Graph API & Library (Gelly), Batch (Table API), Streaming (Table API) Machine Learning (FlinkML)		Malhar for Operators	HDFS Native I/O Codecs with native data compress
Execution Model	Micro-Batch	Event-Guided Stream, Micro-batch	Batch, Streaming	Batch, Micro-Batch	Streaming	Batch and Micro-Batch
Manager	Standalone, YARN, Mesos, EC2	Local, Cluster or Cloud	Standalone, Hadoop YARN, Apache Tez, EC2, GC	YARN	YARN, Mesos	YARN, Mesos
Paradigm	Batch and near real-time	near real-time and real-time	Batch and real-time	near real-time	real-time	near real-time
Partition Management	Yes	No	Yes	Yes	No	No

it is possible to perceive that several languages are used in order to widen the scope of each framework based on the profiles of developers and technologies that are used for both companies and academic solutions. Alongside that, these frameworks bring different libraries of different scopes from necessities of recent applications i.e. Machine Learning. The most popular and which allow stream processing specify different libraries of machine learning, not only that, they also make available in their API the capacity of processing using SQL or graphs e.g. Spark, Storm, Flink.

Furthermore, there are different execution models in these frameworks. The micro-batch concept is that streams can have peaks and troughs. During a lull, the stream flow may not contain enough data to fulfill the batch size in a reasonable period. Batches that are too small can decrease the performance, but quantities that are too large can slow a tuple in reaching its destination (TUDORAN et al., 2014c). Therefore the micro-batch usually comprises the real-time requirements, so it only achieves the near real-time limits. Some applications required fast response, although it may reduce the throughput of events, it guarantees the real time. It is possible to achieve real-time with Flink, Apex and partially in Storm, which reshapes the event-guided stream and eventually comprising the real-time requirements. The stream guided, which is by each events produced and consumed i.e. one-at-a-time (DAS et al., 2014), that approach usually consist in provide the events as fast as possible to achieve the real-time constraints, however it is still possible to encounter the limitation within stream processing system e.g. high latency, bad workload distribution and so on.

The partition management is an important attribute for the processing engines, primarily in the highly distributed system that processes hybrid applications, i.e., stream and batch. It is crucial since the control of partitions is necessary to split the data stream appropriately, assuming that in heterogeneous environments the resources are particularly not equal and the unbounded stream is not continuous and uniform. The partition management is present in some of these frameworks, e.g., Flink, Spark, and Samza, where is possible to swap and define, using the API, the data flow within the cluster environment.

The attributes of each message brokers are compareded in Table 3.2. Comparing Apache Kafka and Apache Flume, it is possible to that the purpose of Apache Kafka is more general since the Apache Flume was initially designed to send data to HDFS and HBase. Apache Flume has specific optimizations for HDFS and integrates with the security of Hadoop. Moreover, it is important to note the lack of fault tolerance in Apache Flume, if an agent fails in Apache Flume, the events will be lost until it can recover the

Table 3.2: Comparative Between Message Queues Frameworks

Framework	**Scalability**	**Data Recovery Failures**	**Protocol**	**Processing Paradigm Supported**	**Partitions**
Apache Flume	Avro, Thrift, Syslog, Netcat	No. When a process of an agent dies cannot be recovered.	JMS Protocol's	Batch processing. It sends data to HDFS and HBase. It has specific optimizations for HDFS and securely integrates Apache Hadoop.	No
RabbitMQ	Yes	Yes. Queues are mirrored to the cluster	MQTT, STOMP, AMQP, HTTP	Stream processing. Slow consumers cause failures in message delivery.	No
Apache Kafka	Yes	Yes. The log partitions are distributed over the Kafka cluster. Each server handling the data and the applications for a part of the partitions. Each partition is replicate through a configurable number of servers for fault tolerance.	Own protocol	Batch and Stream processing	Yes
ActiveMQ	Yes, at certain point.	Yes. If deployed with Apache Zookeper	MQTT, STOMP, AMQP	Stream processing.	No
Mosquitto	No	No	MQTT	Stream processing.	No

disks. Besides, in (KREPS; NARKHEDE; RAO, 2011) is shown that Apache Kafka has better performance both on the consumers as the producers compared to RabbitMQ and ActiveMQ. Thus, allow to manager the batch size in communication in order to achieve higher throughput, this is a functionality which is not presented or introduced by others frameworks.

There are several features in Apache Kafka that is not found in another message queue framework. For instance, for the queues like task queue, resource queue, communication optimization approaches, replication and partitioning. Apache Kafka uses the publisher/subscriber paradigm. Therefore, it creates different topics to control the system. Each topic can be configured with a replication factor, which provides fault tolerance and checkpoints that aiming to maintain safe every piece of data. Alongside the replication, there is also the partition which provides a simple abstraction of multiple queues to a single topic, that runs in parallel increase throughput and reducing latency. It is possible to obtain better performance using Kafka, although it main concern is to guarantee data safety, deliverability and stability which usually speed down its performance.

Apache Kafka demonstrates to be the most appropriate framework for messaging exchange, which meet today's needs of Big Data applications, based on its wide range of features and solutions. From the point that, allows the use of stream and batch within your system, to the guarantees provided by the persistence and consolidation of logs required for stream processing applications.

4 RELATED WORK

Big Data systems require orchestration of tasks, often provided partially by the solutions alongside the presented frameworks platforms. However, these frameworks are concerned only with common cases and still lacks on major points, such as load distribution to its resources.

This chapter will address the aspects of related work closely to the proposal that seek to reduce the gap within the platform that run Big Data. The chapter is divided into four sections: the related work that address the use of dispatcher models in its architecture or use a standard architecture (eg, Lambda Architecture); research that seek to optimize communication in these systems; works that propose different ways of partitioning streams; and finally the most pertinent to this proposal are compared and discussed in the last section of this chapter.

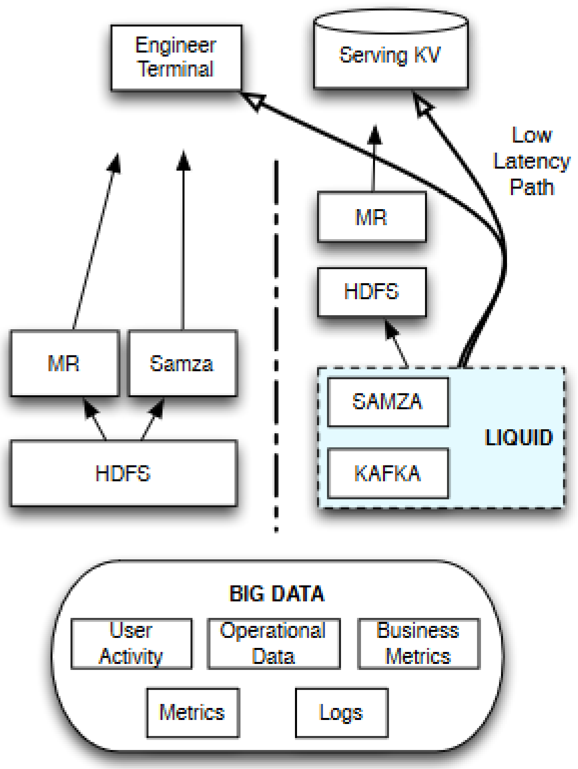
4.1 Dispatcher Models in proposals

Recent proposals are strongly based on architecture; several works propose its architecture or from the classic ones for hybrid processing in heterogeneous environments. Nonetheless, some of these researches presented the dispatching of tasks, on the other hand, most of them assume that the distribution is already ideal and does not provide further details of their dispatcher module. The following subsections are the closest related work, which proposes different dispatcher modules and concerns with the aspects within.

4.1.1 Liquid

The Liquid is a data integration stack which attempts to provide low latency in data access to support near real-time alongside batch applications. It is build to support incremental processing, and cost-efficient and with higher availability (FERNANDEZ et al., 2015). The authors behind Liquid indicate several limitations that affect performance and increase cost during the execution of applications: high latency, principally in write operations in the DFS; the re-processing of data after updates, it is necessary to support incremental processing, since MR/DFS stack lack on that; all jobs consume from the same available resources causing resource interference and compromising the infrastructure.

Figure 4.1: Liquid Stack (FERNANDEZ et al., 2015)

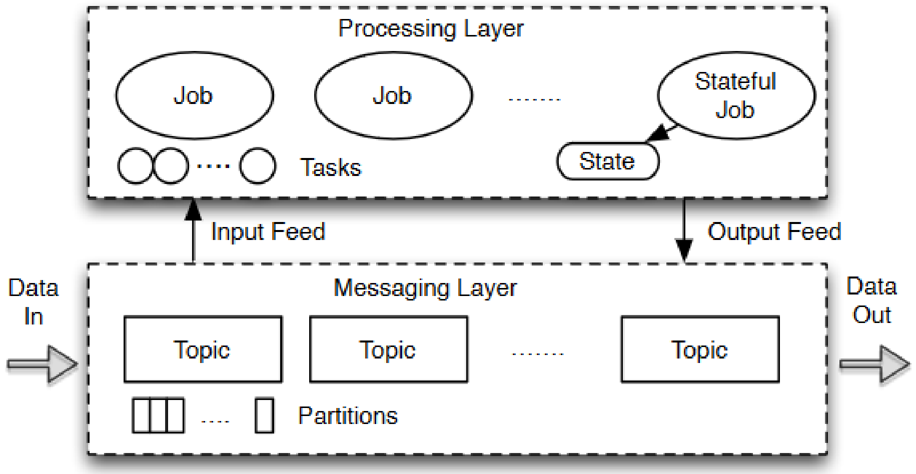


The research argues that is necessary to rethink the architecture model for data integration, therefore proposes a new model, that is not based in MR/DFS model, called Liquid, which can be seen in Figure 4.1. This model is the one used in LinkedIn and has the following properties: Low Latency, which provide fast access to data and also pushes new data to batch-processing; Incremental Processing using annotations with metadata e.g. timestamps or snapshots, which is decisive for incremental processing and failure recovery; High Availability which is data replication of different back-ends in order to provide reliability on cluster failures; Cost Effectiveness a trade-off between high throughput and maintaining low operational cost.

The Liquid model is divided in two layers as can be seen in Figure 4.2, the Processing Layer and the Messaging Layer. The Processing Layer executes the ETL-like jobs for different back-end systems, guarantees service levels through resource isolation, provides low latency and offers rewindability i.e. the possibility to access old data through metadata. The messaging layer is based on topic-based publish/subscribe model that is used as a broker service to persist incoming flow and manage the workload (dispatcher), besides that also persists the data feeds containing the results processed from the Processing Layer.

The dispatching consist of a load-balancing or semantic routing which follows a round-robin fashion or according to a hash function for each producer partition. The consumers from that model are divided into groups, which gives further flexibility in how clients consume data (FERNANDEZ et al., 2015). Inside the groups, only one consumer receives the messages and behaves as a queue. The separation between groups provides a different manner to load-balance the load across the consumers.

Figure 4.2: Liquid model (FERNANDEZ et al., 2015)



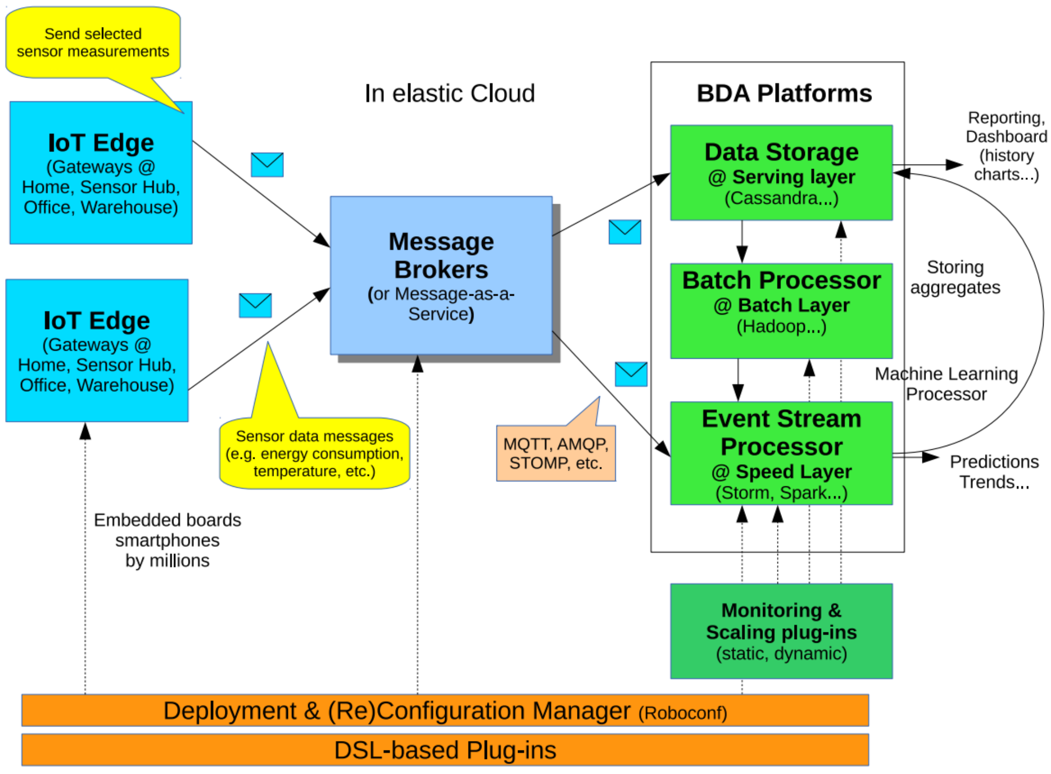
Liquid evaluation was performed using the LinkedIn back-end systems, which is not replaceable over normal circumstances. The deployment of Liquid was done using 5 co-location centers, in different geographical points. These centers ran the messaging layer which is based on Apache Kafka and ingested 50 TB of input data and produces over 250 TB of output data daily. During these experiments was necessary to use around thirty different clusters, comprised of 300 machines that handled 25.000 topics and 200.000 partitions. For the processing layer was used the Apache Samza across 8 clusters with over 60 machines.

During the execution of the experiments was collected operational data, most of metrics, alerts and logs, which are indicated to be essential to react to problems fastly. Besides, it was collected the client-generated- events refereed as real user monitoring in order to analyze and detect anomalies and problems in performance. No further metrics were collect, and no metric was used to evaluate the architecture.

4.1.2 Cirus

Cirus framework (PHAM et al., 2016) proposes a generic, elastic, scalable, re-configurable deployment and multi-cloud framework, focusing on Ubilytics (Ubiquitous Big Data Analytics). The paper evaluates the genericity and elasticity of Cirus through a use case using a set of real datasets. Cirus collects and analyzes IoT data for M2M services (machine-to-machine), which are indicated by the author as producers of vast amounts of data. In addition, scalability and orchestration of the platform are provided through the use of the Roboconf platform (PHAM et al., 2015).

Figure 4.3: Cirus Architecture (PHAM et al., 2016)


Send selected sensor measurements
IoT Edge (Gateways @ Home, Sensor Hub, Office, Warehouse)
IoT Edge (Gateways @ Home, Sensor Hub, Office, Warehouse)
In elastic Cloud
Message Brokers (or Message-as-a-Service)
Sensor data messages (e.g. energy consumption, temperature, etc.)
MQTT, AMQP, STOMP, etc.
Embedded boards smartphones by millions
BDA Platforms
Data Storage @ Serving layer (Cassandra...)
Batch Processor @ Batch Layer (Hadoop...)
Event Stream Processor @ Speed Layer (Storm, Spark...)
Reporting, Dashboard (history charts...)
Storing aggregates
Machine Learning Processor
Predictions Trends...
Monitoring & Scaling plug-ins (static, dynamic)
Deployment & (Re)Configuration Manager (Roboconf)
DSL-based Plug-ins

Three abstract components describe the architecture that is presented in the Figure 4.3; each specialized component is transformed into a concrete component when the application is instantiated in the Cloud environment. These components are represented as layers in the Cirus model, which are IoT edges, message brokers and big data analytics platforms. Also, the BDA platform of Cirus is divided into three main layers (serving, batch and speed) established by the Lambda architecture. Validation is conducted through the deployment of a real Smart-Grid use case, making future predictions of consumption based on real-time data, collected from households, and intends to evaluate the elasticity of the proposal. The prototype consists of OpenHAB as the IoT edge that produces income, Mosquitto as message broker and Apache Spark as the BDA.

4.1.3 JetStream

Stream-processing systems have been explored in literature as a way of collecting, processing, and aggregating data across large numbers of real-time data streams (RENART; DIAZ-MONTES; PARASHAR, 2017). JetStream (TUDORAN et al., 2016) proposes a set of strategies for efficient transfers of events between cloud data-centers. A batch monitoring and self-adjusting platform is proposed to create many connection routes to locate the smallest latency. The authors of Jetstream noticed this loophole and decided to test the effects of replication on the latency between two communication points. First, they measured latency ping between two end-to-end points, then they replicated one of these points in the same data center 3 times. This procedure is performed to determine if the latency between any of the new replica (in the same data center), towards the fixed ending point, achieves smaller latency.

Besides, the dispatcher in JetStream seeks to increase high-performance transfers by batching the events together. Their tests counted with 0 to 1000 events aggregation and measured the latency response of data transfer between North Europe and North US Azure data-centers. They concluded that batch sizes between 10 and 100 units of events are desirable, with a bigger inclination towards 10 units of event aggregation. Still, the experiment achieved transfer rates up to 250 times faster than individual transfer solutions. Also, they achieved 25% additional performance gain in comparison to static batching strategies.

JetStream dispatcher can self-adapt to the conditions of streams by modeling and monitoring a set of context parameters. It further aggregates the available bandwidth by enabling multi-route streaming across the cloud sites. The research focuses on events transfers between inter and intra-nodes. An adaptive Cloud batching where an algorithm aggregates the streams in batches to reduce the latency. The main idea of JetStream is to overcome limitations in the transfers across different topologies of a network. However, it just considers the latency and not the volatility. The proposal focuses on scheduling policies must adapt to the environment.

4.1.4 SMART

The SMART Architecture (SA) proposed by Anjos (ANJOS et al., 2015) considers heterogeneous and geo-distributed data sources, data analysis, consider costs, failure probability, network overhead, I/O throughput and minimize the transfers between the computational resources.

The SA offers an efficient architecture for Big Data analysis for small and medium-sized organizations. The implementation considers heterogeneous systems, varied data sources and geo-distributed environments. Also, are considered the cost, fault tolerance, network overhead, I/O throughput, as well as the minimization of data transfers between computational resources (SILVA et al., 2016). However, these parameters are not enough to work with Stream-processing and heterogeneity. To overcome the environment limitations, it has necessary to input the characteristics from the devices, such as memory, CPU speed and storage. Those information's are important for the load balancing what could be used to maximize the throughput of the Big Data applications.

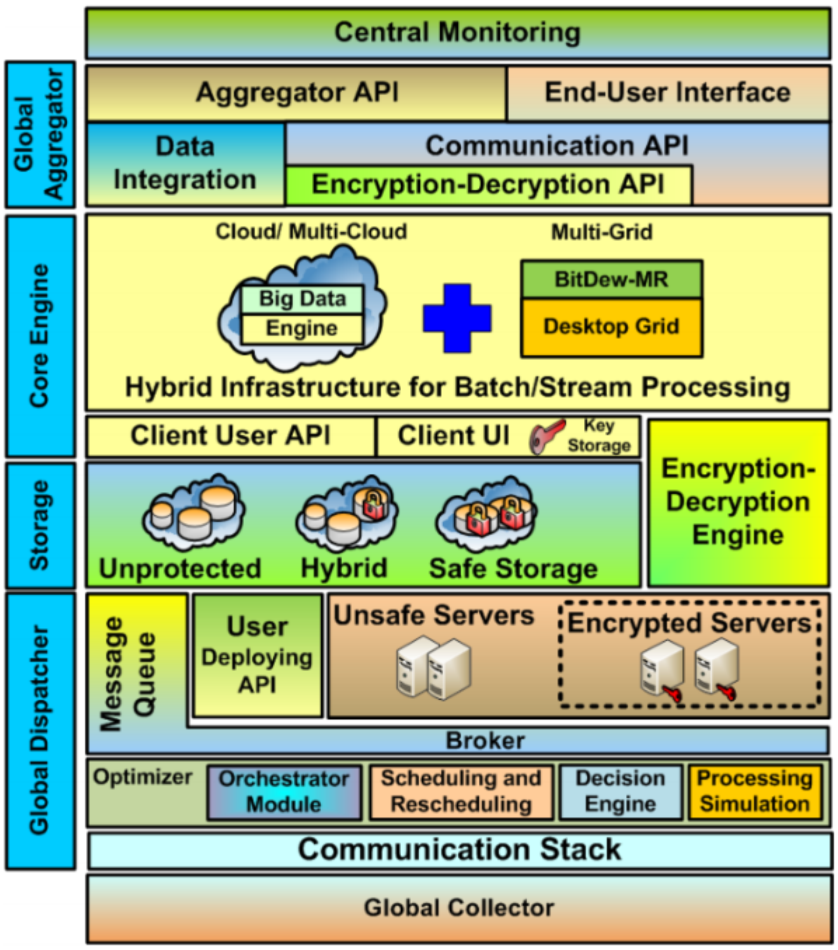
SA offers an efficient deployment of Big Data analysis for small and medium-sized organizations (ANJOS et al., 2015). The complete overview of SA is presented in Figure 4.4.

Based on Figure 4.4 and following the top-down approach, we describe the **Central Monitoring** of SA as a module to monitor distributed systems for both homogeneous and heterogeneous environments, which enables monitoring clouds, grids, and IoT devices. Also, it offers a user-friendly Interface that shows information and insights in real time; The **Global Aggregator** orchestrates data aggregation results and keeps it safe for end-users; **Core Engine** supports hybrid processing such as the provisioning of streaming and batch applications at the same time over Cloud/Multi-Cloud and Multi-Grid environments.

The intermediate results, processed in the Core Engine, are serialized for the Global Aggregator that carries out the data consolidation; The **Storage** layer handles data in protected and unprotected mode. The data are stored in relation or non-relational databases; The **Global Dispatcher** has as objective to decouple data from the lower layers in the message queue mechanism.

The work presented by Julio César in (ANJOS et al., 2015) performed batching experiments in its dispatcher in order to understand the relation between streams rate of income and Flink's processing time fluctuation. In this particular part of his thesis, experiments were developed using Apache ZooKeeper as configuration tool, Apache Kafka

Figure 4.4: SMART Architecture (ANJOS et al., 2015)



as message broker and Apache Flink as stream processing engine on geo-distributed Microsoft Azure(Intel Xeon E5-2670 8 cores @ 2.6 GHz - 56 GB Ram DDR3 -1600 MHz machines). His 20 experiments simulated 30 machines that generated 100 to 100,000 tweets of 2KB each. The results of these experiments showed that the Batch Generation Module does not impact communication because it has low latency and high bandwidth for data transferring. However, the study demonstrates qualitatively that the size of the input affects the execution time in both Kafka and Flink, which shows best performance response to greater blocks of data.

4.1.5 Neptune

Neptune is proposed as a solution to achieve efficient consumption of resources: network IO, CPU, and memory. Neptune provides an intuitive stream processing API alongside a processing graph description model (BUDDHIKA; PALLICKARA, 2016) that describes processing as a collated set of modular stages which is implemented on top of Granules computing framework (PALLICKARA; EKANAYAKE; FOX, 2009). Granules is a lightweight, streaming-based runtime for cloud computing which incorporates support for the Map-Reduce framework; it provides rich lifecycle support for developing scientific applications with support for iterative, periodic and data-driven semantics for individual computations and pipelines.

Granules is responsible by orchestrating the cluster, and it launches tasks at a single physical machine which act as containers for individual computation tasks. Computational tasks are scheduled to run based on a scheduling strategy that can be changed during execution. The scheduling strategy could be data-driven, periodic, count-based or a combination of these.

The key concepts of Neptune are: Stream Packets is the most fine-grained element of data, an ordered unbounded set of stream packets; Stream Sources ingests outside data stream into the engine; Stream Processors apply specific logic to process a stream; Parallelism, which allows parallelizing the stream graph from one instance to multiples instances without dependency; Stream Partitioning, provides schemas for defining how a stream should be partitioned when it is routed to different instances; Stream Processing Graph which is a representation of the classical DAG in stream processing.

Neptune also applies a solution to optimize the throughput; it mainly consists of a leveling buffer and a 'batched scheduling'. The size of these buffers is defined regarding their capacity as opposed the number of messages being buffered. The rationale behind this design decision is to flush the buffer as soon as the required threshold is reached irrespective of the number of the messages in the buffer and their sizes (BUDDHIKA; PALLICKARA, 2016). Besides, it is also applied a batched scheduling which reshapes events sizes in order to optimize communication and therefore increasing the application throughput.

4.2 Communication Optimization

In the case of a scalable environment, the communication plays an important role. In this way, it should be structured to reduce its influence in performance. Since, there is a trade-off between pre-computing and reformatting data, that will influence the transference.

Meanwhile, the unstable internet environment suffers profound changes, it requires to be able to adapt according to computing resources in and out at the platform and the computing resources elasticity must be considered in the network design. For instance, a structured P2P network (i.e. consistent hashing (KARGER et al., 1997)) overcome all the aforementioned problems. Examples as Cassandra (LAKSHMAN; MALIK, 2010) and Twister (EKANAYAKE et al., 2010) show the P2P efficiency.

A study of approaches to determine batch sizes for stream applications is present in (DAS et al., 2014; TUDORAN et al., 2014b) and in Neptune (BUDDHIKA; PALLICKARA, 2016). These approaches commonly loaded the workload (batch-based) before being delivered, so a trade-off is discussed. Also, algorithms were proposed to find the smallest batching interval that ensures system stability for the longest possible period, and that adapts to unstable scenarios. The evaluations were conducted assuming metrics like latency, network throughput and bandwidth, to validate the appropriated size for different scenarios of network conditions, parallelism, operators, and resource contention. Moreover, in NEPTUNE, the solution proposed represents a dispatcher that handles work flow and reshapes data in order to optimize communication, resource utilization and seeks to guarantee real-time or near real-time responses.

In (LI et al., 2016) solutions are introduced to enable elasticity over shared clusters. It mainly focused on the demand where the input rate of streams may vary drastically from each location. A monitor that discovers bottlenecks in the stream data flow is proposed, mostly in stateful operators. In order to solve those bottlenecks, a module is responsible for re-scaling the capacity of processing of each point, by providing additional resources. It seeks to solve the found bottlenecks, and reestablish the data flow over the network. It concentrates principally on the capacity of processing to solve network congestion problems, and not the data itself, as it is proposed in this work.

The author of (DAS et al., 2014) evaluates batching size impact on streaming workloads. In streaming applications, a stable environment can be defined by a predictable high rate of stream income that can be handled by the streaming engine. The opposite of that would be an unstable environment, which might occur sometime between 7-11pm, when cloud providers report high activity. This instability is generated due to a sudden and high increase of stream income.

An algorithm is proposed that attempts to find the smallest batching interval that ensures high throughput and that adapts to more stable scenarios. This algorithm is implemented and divided into modules: Ingestion module or Batching Module (BM), Processing Module (PM) and Control Module (CM). BM receives and treats the income stream flow, while PM runs the application when it is given data. Finally, CM supervises the metrics, calculates the next batching value and informs it to BM. The experiments evaluated aggregation by time, using continuous batching values from 0 to 4 seconds. Also, every experiment ran statically for a given configuration, then all these results were stored in order to achieve a general optimal dynamic response. Thus, the paper started from a static analysis and moved towards a dynamic analysis of batching values, applying however, time instead of a fixed number of events. Their experiments started with no configuration applied to the workload and were able to adapt to changes in cluster resources and stream income ratio. On the other hand, our work seeks to study the impacts of aggregation number per number of events. Furthermore, while both their and our studies used throughput and processing time as metrics, we decided to go further including latency and erasing cache between tests.

The research of Matt Welsh (WELSH; CULLER; BREWER, 2001) presents a dynamic batching strategy that measures throughput and real-time requisites. The work can be applied, amongst other fields, in medical monitoring systems that must provide loyal images in critical time for the sake of the patient. This prototype was implemented with the Batch Controller (BC) as the most notorious structure, because its function is supervising the throughput and the application's response time, both varying in function of the batch size in different stages of the execution. Algorithmically, the values of stream batching may vary from 30 to 100 units of events, decreasing slowly towards the minimum aggregation value that keeps throughput high and processing time low. However, if at any time during execution the system notices frequent drops of throughput for the same aggregation value, it assigns 100 to the aggregation size as the maximum value (100).

Furthermore, it is possible to find researchs that uses the fact that Graphic Processing Unit to Central Processing Unit communication creates event transfer bottleneck, influenced by the GPU greater speed, to justify their streaming algorithm (VERNER; SCHUSTER; SILBERSTEIN, 2011). This way, they researched and discovered, *a posteriori*, that it is much more valuable to assign the complex workloads to the CPU and the less complex to the GPU. Also, since the arrival rate of these less-complex events is greater when compared to complex events, putting more effort towards their completion benefits the streaming system in medium and long-term periods. The GPU is responsible for coordinating the entire prototype, hence, it enlightens one of their main contributions: Using the fast GPU to quickly coordinate their operations.

In addition, their work uses minimum and maximum time variables named deadlines

(DL) that oscillate between 0ms and 100ms. The algorithm's goal is to find an optimal value for the Rectangle, which is a structure proposed by the work that aggregates events in a pipelined way based on DL of their threads. Thus, every thread that is ready to be executed can be aggregated in the Rectangle and sent for execution, diminishing the task's execution time. Based on the DL of the threads, this thread might be assigned towards a more complex or more simple aggregation. This strategy demonstrated a 50% gain in the total amount of processed events in the GPU and CPU configuration when compared against state-of-the-art solutions.

4.3 Stream Partition

Distributed Stream Processing Engines have grown considerably since the ascension of Stream Processing in Big Data scenarios and its necessity of real-time processing. Notwithstanding, the popularity of batch processing is not losing strength, as it can be placed side by side with DSPE (e.g., Lambda Architecture). Lots of data are created day by day, and it comes with a flow, regularly within streams. There are peaks overflows of data produced in different places over several conditions that may lead to the imbalance.

In the research of (NASIR et al., 2017) is presented DSPE as a solution to handle vast volumes of data that come at high rates, and require low latency answers. The authors intend to solve load inbalance caused by skewed streams in heterogeneous clusters. The following main three main solutions to partitioning data streams are discussed: **Key grouping** in which all messages with the same key are processed by the same worker, which is affirmed to be the perfect choice for *stateful* operators; **Partial key grouping** that is adapted from the traditional power of two choices (NASIR et al., 2015) for load balance in map-reduce operators; **Shuffle grouping** that forwards messages blindly, usually in round-robin, it is indicated for *stateless* operators. Finally, a consistent grouping scheme also is proposed, that intends to achieve fair assignment of messages to operators considering skewed streams and heterogeneity.

Furthermore, it is possible to find in the literature different, from simple to complex, approaches to partition data streams (GEDIK, 2014; KATSIFODIMOS; SCHELTER, 2016; NASIR et al., 2017; ATASHKAR; GHADIRI; JOODAKI, 2017; CARDELLINI; NARDELLI; LUZI, 2016). Most of the approaches intend to reduce communication costs and maximize resource usage in a certain level of parallelism. Some of the common approaches are: *Random* partitioning, in which the incoming flow is distributed randomly to each worker following a uniform distribution; *Broadcast* partitioning, which provides the same message to each worker; *Rebalance* partitioning, which partitions elements in an old-fashioned round-robin as it attempts to create equal load per partition.

It is indicated in Apache Flink framework (KATSIFODIMOS; SCHELTER, 2016) as an ideal solution for performance optimization in the presence of data skew; *Rescale* partitioning, which also partitions elements in a round-robin way as in *Rebalance*. However, it does not completely rebalance the data stream. *Rescale* considers the level of parallelism of each worker and produces a subset of downstream operations. The subgroup of operations to which the upstream operation sends elements depends on the degree of parallelism of both the upstream and downstream operation. That being told, the load is split fairly considering the parallelism of the upstream and downstream operations..

There are several approaches for splitting data streams and network optimization, but no pattern is set since there is not a definitive solution to every scenario. It is necessary to evaluate and experiment empirically in order to find a suitable solution.

4.4 Discussion

The dispatcher modules are commonly deployed in environments based on architectures such as Lambda, which provides the adequated platforms to process streaming and batch data. For instance, SMART and Cirus are represented by Lambda, and the dispatcher module is responsible for orchestrating the tasks and managing the workload produced for these two paradigms. In Liquid, however, it is proposed another hybrid platform, which summarizes some of the features found in Lambda into a smaller architecture that achieve the necessities of LinkedIn. Although, the dispatcher module and load balance manage the workload as the fashion round-robin, not applying particular algorithms of partition and scheduling.

Jetstream from another side does not worry with the separation of the tasks and workload; its dispatcher only adapts the messages that navigates through it, aggregating multiples messages and alternating the routes to optimize the transfers within the cluster. Nonetheless, Neptune attempts to achieve efficient consumption of resources using the aggregation of events, using a buffer of messages which is defined statically. Neptune dispatcher is deployed within the Granules, which is the manager of the cluster tasks. It does present promising results like in the JetStream, but those results are only inside the cluster and do not consider the whole environment, whereas the approach could be extended.

Table 4.1: Related Work Comparative

	Criteria					
Proposal	Generic	Elastic	Scalable	Heterogeneous Systems	Communication Optimization	Load Balance
(PHAM et al., 2016)	✓	✓	✓			
(LI et al., 2016)		✓		✓		✓
(ANJOS et al., 2015)			✓	✓	✓	
(TUDORAN et al., 2016)		✓	✓		✓	✓
(DAS et al., 2014)			✓		✓	✓
(BUDDHIKA; PALLICKARA, 2016)	✓		✓		✓	
(NASIR et al., 2017)	✓			✓		✓
(GEDIK, 2014)	✓			✓		✓
(CARDELLINI; NARDELLI; LUZI, 2016)	✓		✓			✓
(TUDORAN et al., 2014b)	✓		✓		✓	
This proposal (Aten)	✓		✓	✓	✓	✓

The Table 4.1 summarizes a benchmark of attributes that are interesting for the resolution of problems that this research attempts to overcome. The criteria base itself on characteristics of dispatcher modules and the solutions alongside it. The proposals were select for this table assuming that, at least, present up three aspects of the criteria to achieve a fair comparison with the existent related works.

The compared related work in Table 4.1 lack in some criteria, such the generic solutions, which is defined by the authors as the solutions that may be applied in varied contexts. The solutions which do not present a generic model usually are indicated only to the architecture, whereas the proposal is developed on, and the specific conditions of the determined scenario. Most of the dispatcher modules can only be applied to the respective architecture which the research was delivered and cannot be replicated in different situations. On the other hand, some proposals focus on the solutions within the dispatcher and allow to be reproduced for different architectures and scenarios, such as the works proposed by: (NASIR et al., 2017), (CARDELLINI; NARDELLI; LUZI, 2016), and in Neptune.

The elasticity is the attribute often used in clouds services, which is the capacity for the resources to scale up and down, according to fluctuating demands (AO; VEITH; BUYYA, 2018). The dispatcher suffers from these changes; therefore some proposals focus on the ability to handle and adapt the elasticity of these environments. Aten does not concentrate on the flexibility of resources, for another side, it attempts to manage and provide a better execution of the currently available resources. However, it is possible to use Zookeeper in the architecture and feed Aten, giving information about the new or removed resources in order to manage new incoming communication and repartition; it is provided further details in Chapter 5.

Commonly the engines that perform hybrid processing provide scalability in its solution, that attribute is required as well in the dispatcher module, which must offer solutions that will scale alongside these frameworks and do not suffer from overhead. The scalability attribute in the Table 4.1 analysis only the dispatcher module. It is possible to perceive the measurement of scalability is vital for these constant growing systems, since most of the proposals focus on providing at least a prof that the solution can escalate in the Big Data level and so on.

In order to process that massive amount of data that is produced the cloud become an alternative to fast deploy of resources and provide volatility to attend the necessities of Big Data systems. Not only that, but the integration using either the public or the private cloud, alongside local clusters is used to suppress the necessity of computation power. Therefore, the heterogeneity of available resources to process data is typical in that scenario and need solutions that can see the resources, not as one, but as multiples instances with different capabilities to provide adequated load balance among the environment.

Most of the present solutions from related work consider the heterogeneity in the system and ensure proper algorithms to balance the workload among the resources. However, during the distribution of workload in these algorithms, most of these proposals do not consider the communication overload that it may generate, or completely lack, or ignore the existence of the overhead of data redistribution. The proposal of this thesis also focuses on that point, whereas there is still space for enhancements.

5 ATEN

This chapter presents the conception of Aten, a dispatcher module for Big Data architectures that use hybrid processing in heterogeneous environments. Aten is built over recent and cutting-edge technology with open source that merges different algorithms from the state of the art in order to overcome common, however hard to solve, problems in hybrid and stream processing systems.

The chapter is divided into three sections, which are the overview of the Aten proposal, the formalization of the proposed dispatcher module called Aten and its components, and finally, provides further details of the implementation aspects and the prototype to evaluate the proposal and to enable the dissemination of its solutions.

5.1 Proposal Overview

Towards addressing communication and data balance efficiently, Aten is modeled as a multi-resource dispatcher for cloud computing system with heterogeneous infrastructures. The spatial distribution of the cloud nodes demands an adequated design of load balance algorithms and event buffering. Notwithstanding, several factors may increase interference in these systems, such as the bandwidth, local resources (CPU, memory) and the spatial distance between the components in the system.

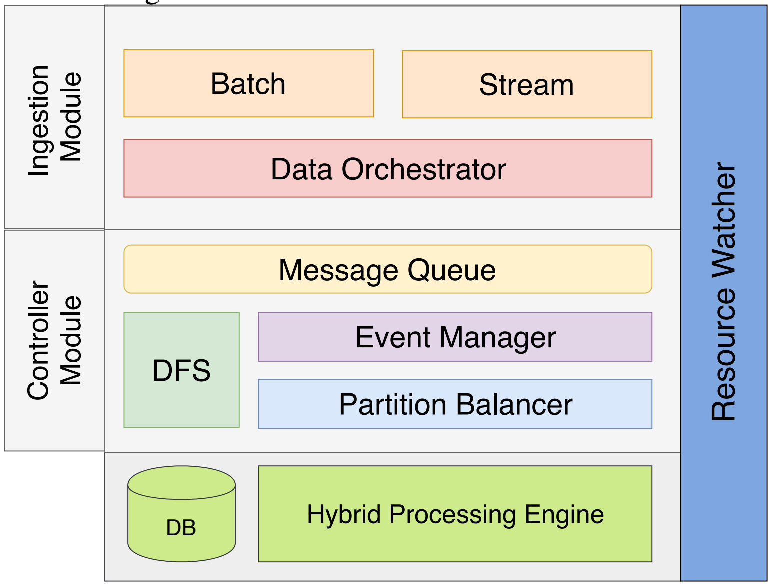
The Aten model is constituted of two main modules, the Ingestion, and the Controller module. The Ingestion Module is responsible for gathering and handling the workload. The Controller Module manages either the batch or the stream workload. In the stream flow, it performs the event aggregations, applies repartition algorithms and provides connectors for the engine. Not less important, also provides connectors for the batch workload with a DFS for further processing.

Thus, the Aten has a layer named resource watcher, which has an overall view and scope of the entire dispatcher and partially of the hybrid processing engine. The resource watcher needs a complete panorama of the modules alongside the engine, and requires a certain level of integration within the engine to help the decision making in the inner components of the modules, and to allow the controller to manage partitions. Further details are provided for each module and the resource watcher layer in the next section.

5.2 Dispatcher Model

In this section, the Aten model is presented towards its modules and the stack of components that assemble it. A representation of the model can be seen in Figure 5.1 proposed by the author.

Figure 5.1: Aten Model. Source: the author



5.2.1 Ingestion Module

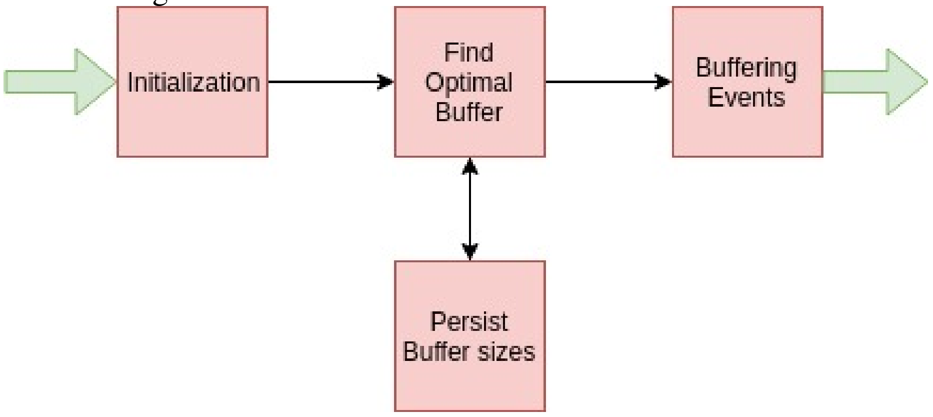
The ingestion module handles the sources of data; it is responsible for establishing the connection between the clients that produce data and the message queue. The clients act in the system as the data producers and are usually distributed in different locations. The clients communicate with the system through requests, therefore sending information connecting using services and sockets. Likewise, the ingestion module regulates the relationships for batch and stream data within the system, separating the requests and forwarding to the proper queues.

The Batch and Stream components work as a service that establishes the connection from the sources, separately. The core of this module is the data orchestrator that manages the stream flow and the batch reception provided by the services of Batch and Stream. The data orchestrator is described in the following subsection.

The data orchestrator implementation applies the buffering algorithm which aggregates events by quantity for stream events. The sequential execution flow is shown in

Figure 5.2. The initialization is the first step within this module, it defines and instantiate the variables and if there is no optimal buffer size selected it executes the function to find the optimal buffer.

Figure 5.2: Data Orchestrator Flow. Source: the author



In order to find the optimal buffer size, the proposed solution takes advantage of the stream application model, which is continuously fed by messages from the data stream. Therefore, we suggest a deadline of 1 second for finding the buffer size. This is the convergence time necessary to the solution find the optimal value, assuming that the application is continually receiving messages to evaluate possible values.

The algorithm initializes using a buffer size of one and increases its size one by one. For every buffer size the duration of the event in the environment are saved in a tuple (*x*, *t*) where *x* is represented by the evaluated buffer size and *t* represents the average time found for the respective buffer size. Therefore, during the convergence time of the algorithm a vector V is filled with the tuples and it is persisted in memory. The final step is to found the minimum ratio between buffer size and time inside the tuples and select the respective buffer size. It is given the limit Equation 5.1 following a linear convergence λ which is the value ratio assuming $\lambda \in (0,1)$ and diverges when $\lambda > 1$.

$$\lim_{x \to 0} \frac{t_1}{x} = \lambda \tag{5.1}$$

Once the definition of buffer size is found the next algorithm follows the optimal value selection. The algorithm for aggregating the events into a single one is described as follows.

The pseudo-code of the implementation is presented in the Algorithm 1, every client runs the application, and the input is the event which is being sent. It overrides the common send method for the events inside the application client, applying the approach. The algorithm buffers events in memory matching the criteria in order to increase throughput, optimizing the use of the available network.

Algorithm 1 Send Aggregated Events

```
1: procedure SEND($event$)
2:     $eventLen \leftarrow lengthofevent$
3:     if $isBatch(eventLen)$ then
4:         $send_batch(event)$
5:         return $True$
6:     $state.put(event.id)$
7:     $i \leftarrow state.size()$
8:     $buffer.append(event)$
9:     if $i >= GetMaxBufferSize()$ then
10:        $send_event(buffer)$
11:        $state.clean()$
12:    return $True$
```

This algorithm is applied for either stream and batch, however only executes the aggregation for the stream events. Furthermore, it persists the state of the events that are being buffered, in order to provide a certain level of fault tolerance in case of the unavailability of the dispatcher services, this enables the possibility to identify events that somehow were lost and re-send them.

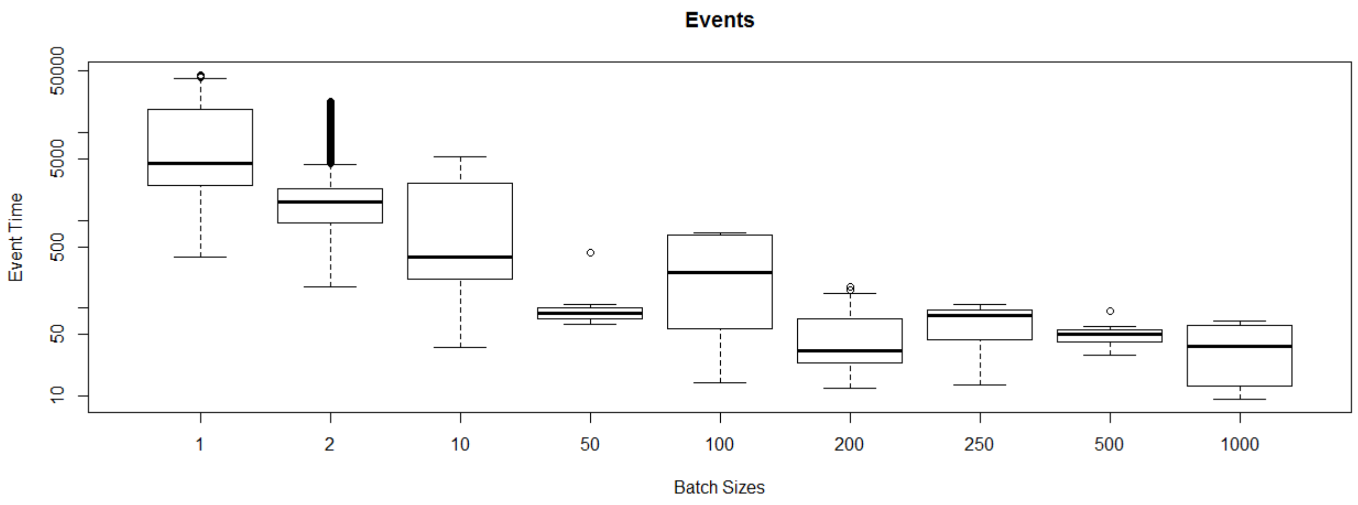
Initially, the length of the event is evaluated in order to identify if the produced event is a batch file or a stream event. If the event is a batch file, it is forward for the batch queue, and if the event is a stream event, firstly the id of the event is preserved. The identification of the event is persisted in order to enable the reproduction of the aggregated events in case of transmission failures. After that, the event is appended to the buffer, which holds all aggregated events. When the buffer achieves the maximum buffer size, the application performs a single send of all aggregated events. The *send_event* method guarantees at least once the delivery of the message, in case of failure an exception is raised and the content of the buffer is restored using the persisted state.

The condition to send the messages in the Algorithm 1 is the number of events that are produced, which brings some considerations that need to be taken into account. The comprise of real-time constraints and requirements of the application are usually subjective. The algorithms to send messages works as a window that depends on the next event processed to send the others messages, in environments that somehow does not have a continuous flow, it will result in events delay, therefore the tradeoff between increase the throughput of events and the real-time constraints must be evaluated considering the application. However, some others conditions could be applied to that algorithm. Aten provides in the configuration a conception close to the time window provided by the stream processing frameworks.

In (TUDORAN et al., 2014b) a discussion was started on the different conditions that could be used to buffer events. The Aten dispatcher provides the following criteria to aggregate events: aggregation by time, which creates a window that buffers the events until the window reaches its deadline and then sends the events within; and combining the time window with the number of events, in case of the buffer does not achieve the quantity to grant the condition, the time window will delivery the events. The prototype described in the next section present the implementation which uses the aggregation by quantity of events.

Now, the definition of buffer size is not an easy task. Several properties define the behavior of the application on the internet, and in a virtualized system, therefore, it impacts also in the size of the buffer of events, that can easily saturate its enhancements. Towards the instability that is found in these systems, initials experiments were performed to evaluate the behavior of the aggregation algorithm. Thus, the configuration, design, and description from that experiment is available in the appendix.

Figure 5.3: Event average duration in ms



The variation of buffer size (i.e., aggregation size) was evaluated in a stream processing environment, the metrics was the duration of each event using different buffer sizes, and later it was estimated in different environments with the interference of latency. The Figure 5.3 show the box plot with the initial results, the batch size of one only forwards the events, which is commonly known as one-at-a-time (TUDORAN et al., 2014b).

A behavior was found within the definition of the buffer size, as could be seen in the figure, it performs like a tendency curve which the module attempts to achieve the minimum point of that curve. It represents the ideal size for the communication, which is the convergence point of the algorithm to define the buffer size. However, there is some considerations that require discussion, for instance, as the buffer size increases the real-time constraints defers, since it holds the stream flow to aggregate the events into a bigger message.

Thus, it is possible to see the positive impact that the algorithm provides even in a configuration that aggregates two-at-a-time. There is considerable variability in low size buffers; the explanation behind it is the more significant quantity of requests and packages sent over the network, more susceptible to packet loss and infrastructure noise. As the buffer size increases, the time to process an event reduces, and it provides higher stability. Therefore, the algorithm attempts to maximize the throughput, and minimize the event duration time finding the ideal buffer size.

Finally, after the Data Orchestrator setup the clients and defined the attributes for the algorithms, it connects the clients with the instances of the Message Queue. The Message Queue is presented as a generic component that can be placed as the Apache Kafka or the RabbitMQ broker. So, the Data Orchestrator creates the topics, the queues, or exchanges in the generic message queue and start to publish the events and messages. From that point, when the data is made available in the message queue, the Controller Module starts its iteration inside the dispatcher. -

5.2.2 Controller Module

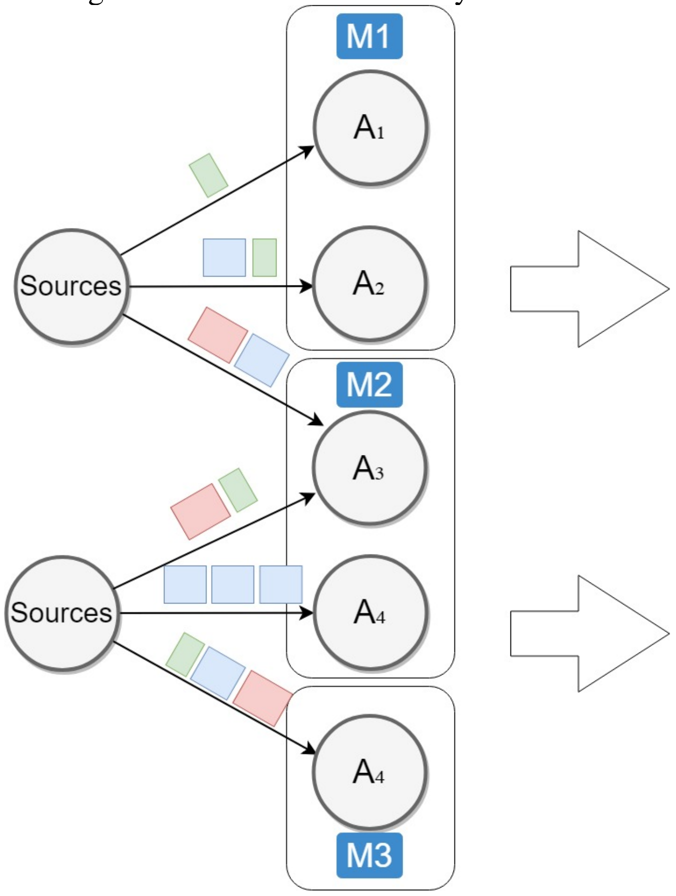
The controller module contains the message queue, the event manager, the partition balancer and the DFS. These components assemble the inner core of the dispatcher. The event manager identifies the events and set labels in the events, and the batch is dispatched for the distributed file system, it allows the application to process the data when it is available inside a shared location. The events identified as stream data got their partition inside the message queue tracked and forwarded to the partition balancer.

The partition balancer splits the incoming data from the stream flow, and this process is represented in Figure 5.4. The circles represent the operators inside the stream processing application, and the rectangles define the machines in which the operator is placed. The boxes represent the incoming data over the lines, which is split and put for the operators. Besides, the partition balancer only applies the load balance algorithm for distributing the data for the machines and does not comprise the behavior inside the cluster. Of course, the initial decision of the load balance will reflect directly in every step of the application inside the processing engine cluster. Therefore, showing that is necessary to have an adequate algorithm which partition the data.

Thus, it is important to point that in a heterogeneous environment the machines have different resource capabilities requiring load balance algorithm, principally for skewed data production inside stream processing systems.

Hence, the partition balancer module is responsible for applying the load balance algorithm. This current research proposes, implements, evaluate and make available for Aten distinct algorithms for load balance in stream processing systems, such as broadcast, naive, random, rebalance and rescale. Every algorithm is discussed and presented in the following sections.

Figure 5.4: Stream Partition. By the author.



Besides, the partition balancer profiles the workload from the partitions and applies the repartition algorithm, and it connects the hybrid processing engine parallelism in each of the available partition. It maps the available partitions inside the message queue and distributes the tasks for the operators inside the hybrid engine. It takes into account criteria, such as the level of parallelism of each operator, the type of the workload or the custom algorithm that is being used to the physical partition inside the cluster. The standard algorithm to the partitioning is the traditional round-robin, in Aten, it is evaluated a series of implementations, from state of the art or from the frameworks that already allow the management of partitioning.

The available algorithms that the partition balancer implements are: the naive partitioner which is proposed by the author, it consists of diffuse events with a bigger length for the partitions with a higher throughput of events. Not only this but furthermore, also connect multiples partition to a single output instance; the broadcast algorithm which distributes all events to every resource inside the cluster; the random algorithm which randomly select one output channel destination between the partitions; and the rescale and rebalance that were implemented based in the proposal of (KATSIFODIMOS; SCHELTER, 2016). Thus, the naive algorithm is feed by the resource watcher, which uses monitoring tools to check and identify the throughput of events inside each instance from the cluster of the hybrid engine.

The resource watcher is implemented as a service, and it runs inside the message queue cluster, specifically in a single machine (the master node). It only persists the data from the resources for a window of five minutes which is being fed by a monitoring tool from the hybrid processing engine, every second. No historical data is necessary to provide insight for the partition balancer module; therefore it attempts not to overload the usage of resources e.g., memory, CPU in order not to cause overhead in the application running in the master machine.

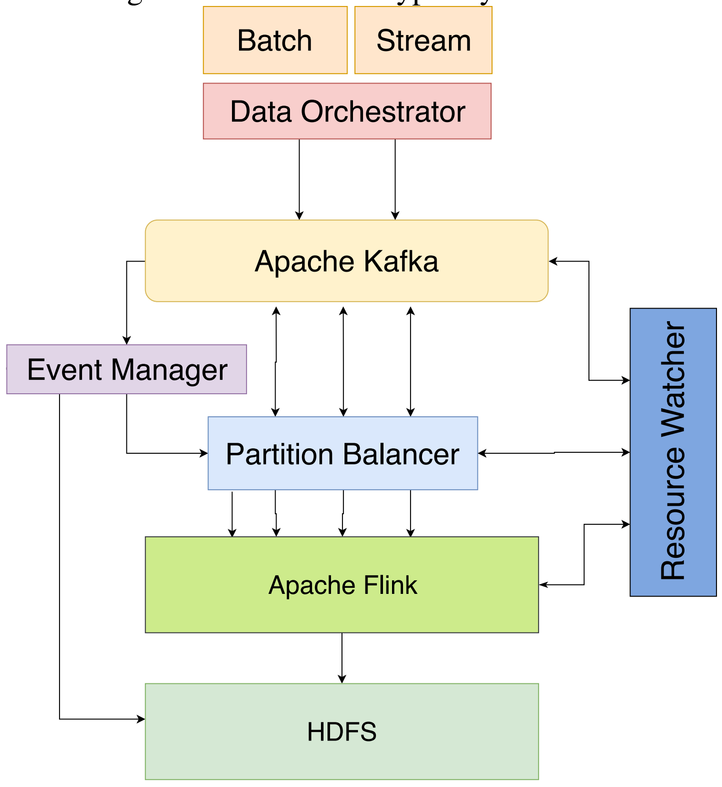
The naive algorithm receives that name since its implementation iterates over the nodes identifying the throughput of every machine and distributing the bigger messages to the machines with proportionally greater throughput. Initially the algorithm does not have enough information to have a proper decision, so it randomly distribute the data among the nodes. After that, it starts to making decision assuming the available logs which the resource monitor collects at every second. It is naive since when it achieves the maximum throughput of events, it considers that it is the ideal distribution of load inside the cluster. Furthermore, in a volatile environment the solution may not be ideal, since when a slave nodes join the cluster it will receive less workload, since it will have no throughput, and therefore its capacity will not be explored.

The rescale algorithm depends on the degree of parallelism of both the upstream and downstream operation, if the upstream operation has parallelism two and the downstream operation has parallelism four, then one upstream operation would distribute elements to two downstream operations while the other upstream operation would distribute to the other two downstream operations. If, on the other hand, the downstream operation has parallelism two while the upstream operation has parallelism four then two upstream operations will distribute to one downstream operation while the other two upstream operations will distribute to the other downstream operations. The rebalance distributes the data equally by cycling through the output partitions verifying the buffering queue of events inside of each resource.

5.3 Prototype

In order to validate the proposed model a prototype is developed; the software stack of the prototype is shown in Figure 5.5. The main language which the prototype is implemented is Python, but the prototype is also integrated with bash scripts and Java.

Figure 5.5: Aten Prototype. By the author



The entire ingestion module is developed using Python 3 and uses open source libraries to open connection with the clients and the message brokers. The stream and batch clients behave as the data source, producing data continuously. The data orchestrator provides the ideal buffer size for the clients to use in its transfers, the value is defined empirically.

The message queue in the prototype is the Apache Kafka 1.0.0, since it provides a high level API, scalability, system logs, replication and repartitions. It can handle either stream events and batch data, which is adequated for the proposal. The API allows to manage the partitions of each topic and setting up the consumption of the data for the partition manager.

The hybrid processing engine used is the Apache Flink 1.1.5 with Hadoop 2.4, which has shown an efficient solution to process batch and stream, providing further enhancements in data distribution and management. The Flink application is written in Java and there are some integration with the dispatcher (event manager). It first establishes connection with the dispatcher, that returns the properly queue that the data must be consumed. And also, the event manager interfaces with the DFS, where the prototype uses the HDFS, to store and put the incoming batches in a distributed file system path folder.

The system provides the Java$^{\text{TM}}$ Platform, Standard Edition Development Kit (JDK$^{\text{TM}}$) 8u181, which is used by the core of Apache Flink and Kafka framework. It also comprises the JMX framework and the serialization schemas that were used in the prototype.

The partition balancer is partially written and Python, and in Java 8, it is required to be in Java since the Flink API does not offer the physical management of partitions in another language than Java. The partition balancer is configurable in the deploy of the dispatcher and applies the algorithms of partitioning. It is statically defined and requires to redeploy to change the algorithm. Although in Flink, it is possible to use multiple algorithms to partition data streams, inside the cluster in inner operators of the DAG.

The resource watcher consumes the logs files from the Apache Flink monitor, using the JMX monitor, like a service that provides insight about the states and conditions of the cluster (i.e., throughput, memory consumption, cpu average, ...). The resource watcher is written in Python and performs as a client which consumes information from the JMX monitor, but also performs as a server, since provide services which allow get in formations about the status of the cluster in order to make decisions.

6 EVALUATION AND RESULTS

The evaluation and experiments conducted of the prototype is presented in this chapter. The chapter is divided in three main sections: the experimental environment deployed to evaluate the prototype; the methodology that was used in order to validate the prototype; and finally the final results obtained from the experiments metrics, which were separated in three subsections. The final section also provides an analysis from the results and discussed individually the Aten possible setups. All the evaluation is conducted in order to demonstrate the functionality and effectiveness of Aten dispatching events in Big Data applications.

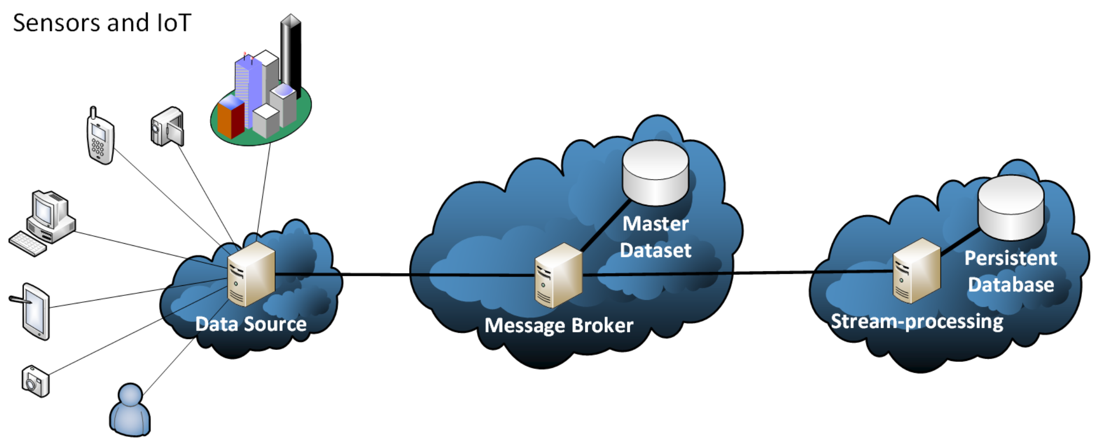
6.1 Environment

The experiments were performed in a cloud environment, assuming the benefits that the cloud computing offers, and also the fact that most of the validations from state of the art are conduct in cloud environment (TUDORAN et al., 2016). Some experiments of the state-of-the-art were performed from the private cloud to the public cloud, usually in Azure, Amazon or Google Cloud. The experiments were performed with Microsoft Azure Cloud Computing Platform & Services in distributed data centers. The created setup intended to analyze the viability of the solution on multi-cloud environments (i.e., multi-tenancy, resource consumption, application behavior, latency).

All experiments were performed selecting Virtual Machines (VM) provided by Microsoft Azure. Different types of VM were selected to provide a level of heterogeneity. However, the producers are proportionally equal in resource capacity, but the data generation rates may be different. This is necessary to provoke the skewed flow of messages in the environment. The dataset is composed of *tweets* previously collected using Twitter API. All *tweets* were filtered using the last US election as the subject. The dataset has 10 GB of tweet data, and each machine receives a portion of the data "2GB slice". The data slice distribution is the same for every client (thread). Inside the producers, the threads are created in order to saturate the available network and achieve a producing rate close to real Big Data scenarios.

The distribution of the experimental platform is illustrated in Figure 6.1, which is made in three levels: the data producer using 10 A3 instances in which each instance produces data as 100 clients (threads) in different ratios, the client applications is implemented

Figure 6.1: Experimental Platform



in Python, and it is located in a Microsoft Azure Datacenter in Brazil; The broker concerns the temporary storage of the messages to be later consumed by the processing engine. It is distributed in 3 A4 Basic VM instances that are managed by Zookeeper and runs Apache Kafka as message queue framework, located in the east US data center. Finally, the processing engine is composed by 4 VMs instances with different resource capabilities: A8 (i.e., f-master), D11 (i.e., f-slave1), A4 (i.e., f-slave2) and A2 (i.e., f-slave3), located in the west US data center. The details of each virtual machine instance is shown in the Table 6.1.

Table 6.1: Microsoft Azure Instances Details

Size	CPU Cores	Memory	(Max Disk Sizes / Available)	Max IOPS
A2	2	3.5 GB	OS–1023 GiB, 40 GiB	2000
A3	4	7 GB	OS–1023 GiB, 80 GiB	4000
A4	8	14 GB	OS–1023 GiB, 100 GiB	8000
A8	8	56 GB	OS–1023 GiB, 1023 GiB	8000
D11	2	14 GB	OS–1023 GiB, 200 GB	2000

The disproportionate distribution and resource capability is necessary to achieve heterogeneity of resources. In the software layer, the framework solution for Big Data hybrid processing used is Apache Flink, which run a common word count application that consolidates the results and reduces data alongside big batches. Every VM instance has the Intel Xeon E52670 2.6GHz processor, DDR31600MHz RAM and the Operational system is Ubuntu Server 16.04, also the virtual machines have a 1,000 Mbps limit of bandwidth, that limit applies whether the outbound traffic is destined for another virtual machine in the same virtual network, or outside of Azure.

The broker cluster is configured with two principal queues (topics), one to persist the data from stream events and the another for the batch files. There is no replication for these topics, since the evaluation does not consider the possible failures inside the cluster. However, the data stream queue is separated in 8 partitions, the batch queue has only 2 partitions since the research focuses in the distribution of event streams, therefore the batch distribution is not evaluated, just forward for HDFS.

The engine cluster is composed of four robust machines, focusing on CPU processing and availability of memory. Apache Flink is deployed with one job manager responsible for managing the task managers, a total of four distributed task managers. Each task manager has a level of parallelism, and the application is distributed fitting the availability of cores in the machine. Therefore, some operators inside the application can achieve a higher throughput rate, assuming different parallelism available in the machines. Thus, the benchmarked application running in the engine cluster is a hybrid word count application which processes new incoming data (tweets) and cross apply functions with the batchs stored in HDFS.

6.2 Methodology

Experiments were defined based on the methodology of evaluation using designs patterns proposed by (JAIN, 1991). The design consists of every possible experimental scenario using the different variables that are evaluated. The variables are defined by the configurations that the dispatcher provides: the presence or absence of the buffering approach and the partition algorithms that the dispatcher implements: broadcast, naive, random, rebalance, rescale, and the baseline execution which is the default partition algorithm present in Apache Flink.

Furthermore, assuming our variables, there are six different algorithms and two buffering scenarios, using a replication value of 30. It leads us to a total of 360 experimentation scenarios. The main comparison is the baseline scenario without the buffering whilst the absence of functionalities from the dispatcher. It is still necessary to use the dispatcher to maintain the "proximity" of the test to achieve a fair comparison; therefore the modules Data Orchestrator and Partition Manager are disabled to provide the default scenario.

However, it is also compared the baseline and partition algorithms not using the buffering provided by the data orchestrator and event manager. We believe that is ideal to ensure that insight since in cases that the real-time requirements are crucial, the one-at-a-time can be enhanced by the partitions algorithms in order to provide better distribution of data and high throughput.

In order to measure the results metrics are defined. The metrics used are from the common metrics used from measuring the stream processing systems and load balance algorithms; the metrics are: the event duration, which is the total duration of an event to be entirely processed; the application throughput per node; and the resource consumption for CPU, memory and bandwidth.

Thus, we attempt to reach the same conditions to every experiment. At every execution of an experiment, the cache is cleansed, and the environment is reset and reinstated. In case of failure in any step of the architecture or the operators, the register is deleted and reprocessed.

6.3 Results

In order to collect the results, some approaches were adopted. To obtain the lifetime of an event, it was logged in the message a timestamp that identifies each step of the event. Using that, it was possible to measure the overall event duration. Besides, in the resource watcher module were coupled bash scripts, it allows to estimate the resource usage of each machine that the dispatcher acts. The shell script run the dstat[1] application which is a versatile tool for generating system resource statistics.

Furthermore, the JMX monitor was used to measure the throughput of events inside the framework. The JMX is a Java framework for monitoring java applications, and it provides easy integration with multiples systems and big data frameworks from the Apache family. The next following subsections discuss the results collected of each metric.

6.3.1 Events Duration

The event duration can be discretized into several stages, however, in our evaluation, the measure of event duration only takes into account the duration of the event serializations, transfer and processing time. Therefore the formula used to measure the event duration is the following:

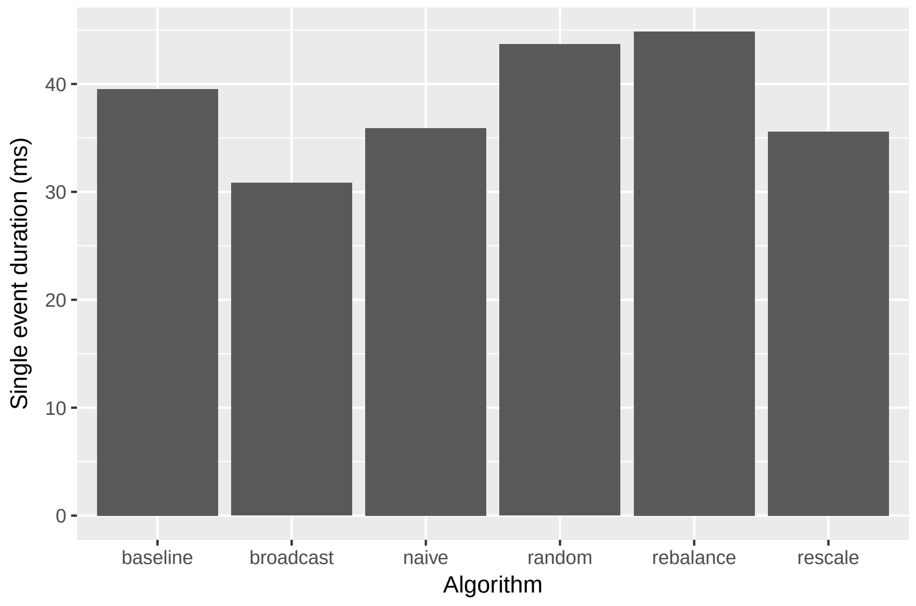
$$D = \frac{1}{n} * \sum_{i=1}^{n} (d_{\mathrm{i}} + s_{\mathrm{i}} + p_{\mathrm{i}}) \tag{6.1}$$

The event duration metric was collected by associating the difference within the

[1]Avaialble in https://linux.die.net/man/1/dstat

timestamps captured in each expected step. Where D is the total average duration of an event, d represents the time duration of the network transfer, s represents the overall serialization time of the events and p is the processing time duration inside the processing engine. The buffering techniques overhead is not separately evaluated since it is insignificant, and the benefits from the adoption of this part of the model only provide gains.

Figure 6.2: Single event duration in ms

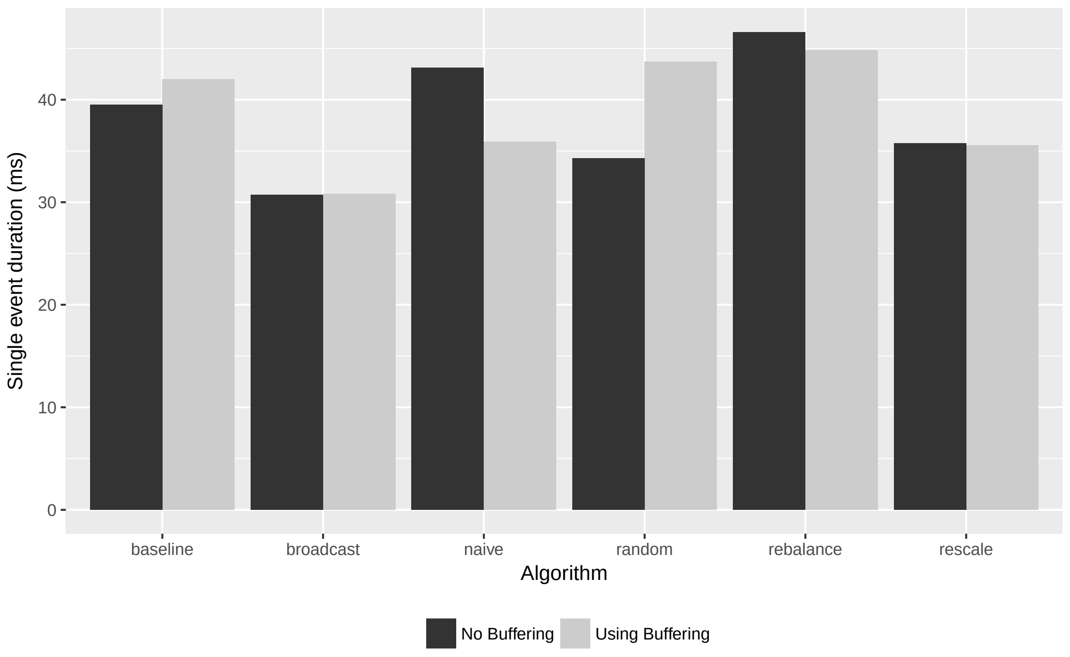


The Figure 6.2 present the overall duration of a single event using different partition algorithms. First, we need to pinpoint that, the algorithms except the baseline do not carry a single event, since the dispatcher aggregate events into a single one, and in the baseline, we use only one event to provide a fair comparison of the model. Therefore, looks like the duration of a single event of the baseline algorithms perform as good as every partition algorithm, which is not the case, since the other algorithms pack at least 128 events alongside a single event.

Still in Figure 6.2, we can see that broadcast performs the better event duration, that may be explained since the whole data is available in every machine, so there is no dependency to process a result, providing a fast response. Rescale followed by naive also performs really well among the others, but still too close to each other to prove a real gain. Thus, random and rebalance performs poorly, except from the baseline which took almost 40 ms to process only an event without any buffering.

Furthermore, it is presented in Figure 6.3 the execution per buffering adoption. As could be seen, the baseline did not perform well against the others partition algorithms in terms of distribution, but the baseline can still be enhanced as showed in Figure 6.3,

Figure 6.3: Single event duration in ms



since it buffers over 128 events inside a single one with an overhead of 3 ms. Thus, the naive application provides a lower event duration using the buffering which seems ideal to combine this algorithm alongside the buffering approach. For another side, the random algorithms have the more significant time duration growth among the other algorithms. Moreover, the broadcast and rescale almost do not provide any overhead for using the buffer approach.

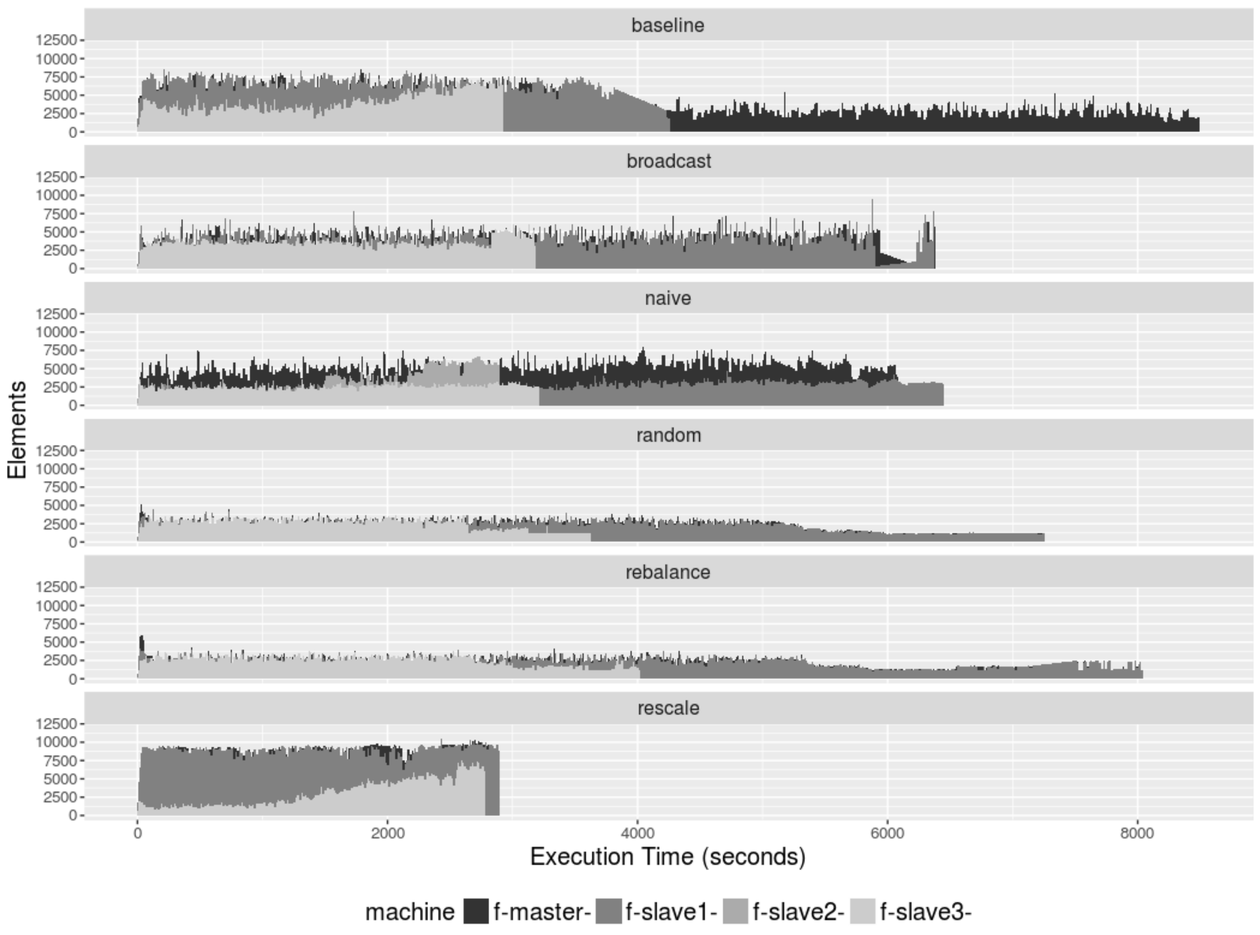
6.3.2 Events Throughput

It is common to measure stream processing applications by its throughput, and inside a single application, there are multiples throughput ratios. Thus, only the operator that has the limiting performance will dictate the overall throughput, and it is traditionally called bottleneck (AKIDAU et al., 2015).

The event throughput measured in these experiments were from the sink operator, which is usually the final step of an application. It provides the overall insight about the processing time of the events, this is ideal, since the idea is to evaluate the distribution and the communication improvements that the dispatcher may provide in the whole process.

In Figure 6.4 it is shown the throughput of events from the application for each partition algorithm and the performance of each node from the engine cluster. The baseline is this figure uses no buffering and partition algorithms, and this is necessary to compare

Figure 6.4: Throughput per node

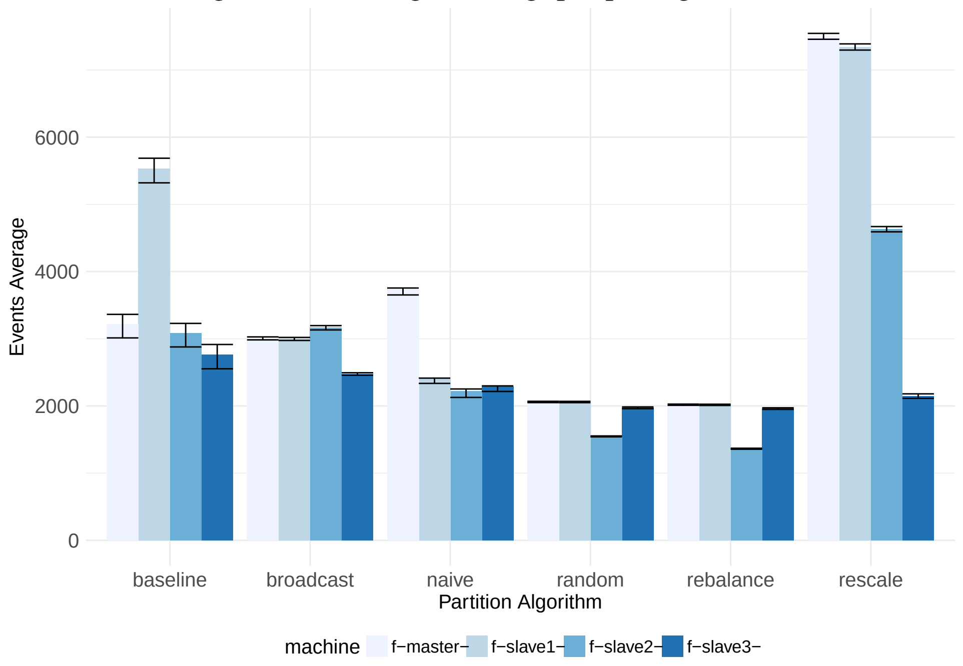


the impact that the use of the dispatcher can achieve. Baseline starts performing well, however during the half of the executing it dislocates the usage of the slaves to complete the final operations only in the master node. This behavior is not ideal since resources are idle and the results are delayed, for almost the double of the time it can be computed using a partitioning algorithm.

Random and Rebalance perform not well, it has a poor distribution that cannot achieve a high throughput compared to other approaches, even the baseline performs better initially. The broadcast algorithms perform better since it distributes date to each node making use of every available resource in order to achieve a result. That seems ideal, however, much unnecessary effort is made to process the data, this can be seen during the end of the execution where it the memory overflows and requires the recovery of the processing engine to finish to process the whole dataset.

The naive algorithm provides a close to fair distribution, although the throughput is not as high as the rescale and baseline. That can be explained since there is a few overhead in reading the partition of higher performance to reassign the data flow. Still, naive performs better than the broadcast algorithms that use an even more naiver algorithm. Of course, it could not reach the results found in rescale, which distribute the data flow for the operators with more parallelism. The slave3 starts with really low throughput since its parallelism is the weakest among the machines, and increases over the execution since the others machines reach its max throughput.

Figure 6.5: Average Throughput per algorithm

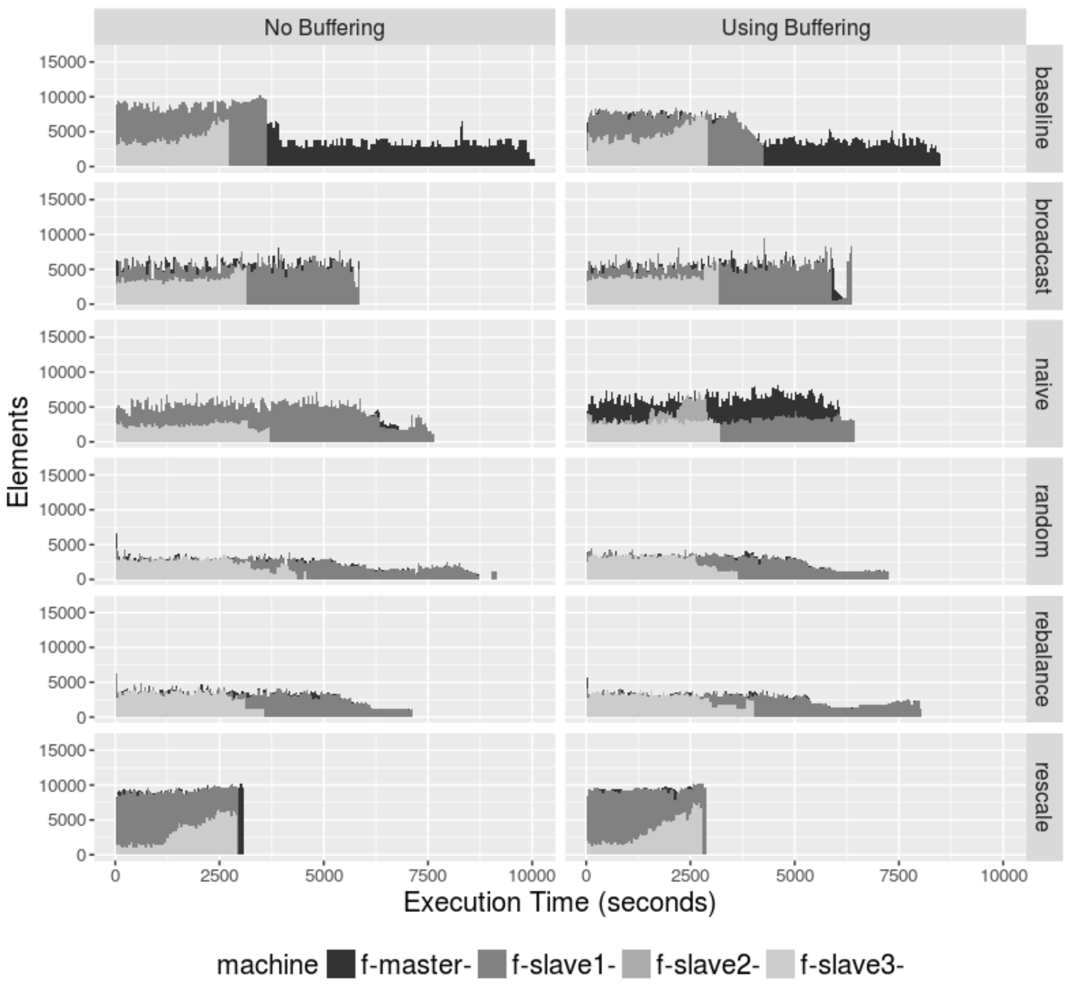


The Figure 6.5 pinpoints the average throughput that the application could reach per second during each experiment. In order to provide a fine-grained measure, the data was measured in 100 ms; however, the number of logs produced and the significant variation of the result was not visually able to be shown and provide some insight. Therefore, the grain size was increased to 1 seconds to ease the visualization.

Still in Figure 6.5 all the results shown are using the completing setup of the dispatcher. The baseline uses no partition algorithm but uses the buffering of events. It performs significantly well comparing to the broadcast, naive, random and rebalance. However, the error is considerably higher than the other algorithms. It achieves an excellent throughput, but the execution continues only in the master as could be seen in the last figure; therefore it performs become unfeasible in resource exploitation.

The naive algorithm throughput perform as good as the broadcast, considering that overload of tasks in the master node, still it distributed somehow equally among the machines, which might be not ideal for the heterogeneous scenario. The performance of rescale is unquestionable, it achieved the higher throughput for each resource, even the broadcast algorithm which has a complete distribution lacks in performance when the matter is the throughput of events.

Figure 6.6: All scenarios - Throughput per node



In Figure 6.6 are present all scenarios throughput - applying and not applying the buffer approach over different partition algorithms. The idea is to show the impact that the buffer creates in the partition algorithm and how they are intrinsically connected. For instance, the baseline suffers from some enhancements when using buffering, since it total execution decreases 15.44%. Most of the algorithms with no buffering are outperformed by using buffering, which it is not the case of the rebalance and broadcast which have a peculiar distribution and its time to process the data increased.

In Table 6.2 is presented the average duration time of each experiment and the difference between using and not using the buffering approach. It illustrates clearly the positive impact that the buffering provides, and the random algorithm can achieve at least 20% more efficiency using the buffering. Followed by the naive, baseline and the rescale, on the other hand, it is clear that it does not help the broadcast and the rebalance. Thus,

Table 6.2: Total duration time in seconds

	No Buffering	Using Buffering	Difference (%)
Baseline	10052	8499	15.44
Broadcast	5859	6384	-8.96
Naive	7665	6445	15.91
Random	9153	7258	20.70
Rebalance	7124	8042	-12.88
Rescale	3080	2895	6.00

the rescale presets the better distribution of load for the resources, and achieves the more significant throughput, thus it can still be optimized using the buffering.

6.3.3 Resource Consumption

The resource consumption was also measured in order to provide further analysis from the behavior of the dispatcher. The CPU, memory, and network in and out usage were collected and are presented in this subsection. The results are the average found between every replication inside the design of the experiment, every result meet 95% confidence interval.

Figure 6.7 illustrates the average CPU usage for each machine from the cluster. It compares side by side the presence of the buffer approach, for each partition algorithm. Overall the performance regarding most CPU consumption the buffering criteria explores significantly more the available resources, not the case of the rebalance and partially the broadcast. The random and the naive algorithm shows the greater exploitation of the available resources, although the last results do seem to achieve greater enchantments for the application.

Notwithstanding, the memory is an essential factor for the big data processing since to compute the raw data is necessary to persist it in memory. Thus, the volatile memory usually costs more than the disks to store it, so it is common to have limited memory space to transform the data.

In Figure 6.8 is shown the average memory usage of each node. There are few contrasts in this figure since the framework has an optimized use of the memory and is able to manage it properly.

Still, it is possible to see some picks of memory usage for the buffering approach, either for more and less usage, e.g., the baseline which explores more the usage of memory

Figure 6.7: Average CPU Usage per node

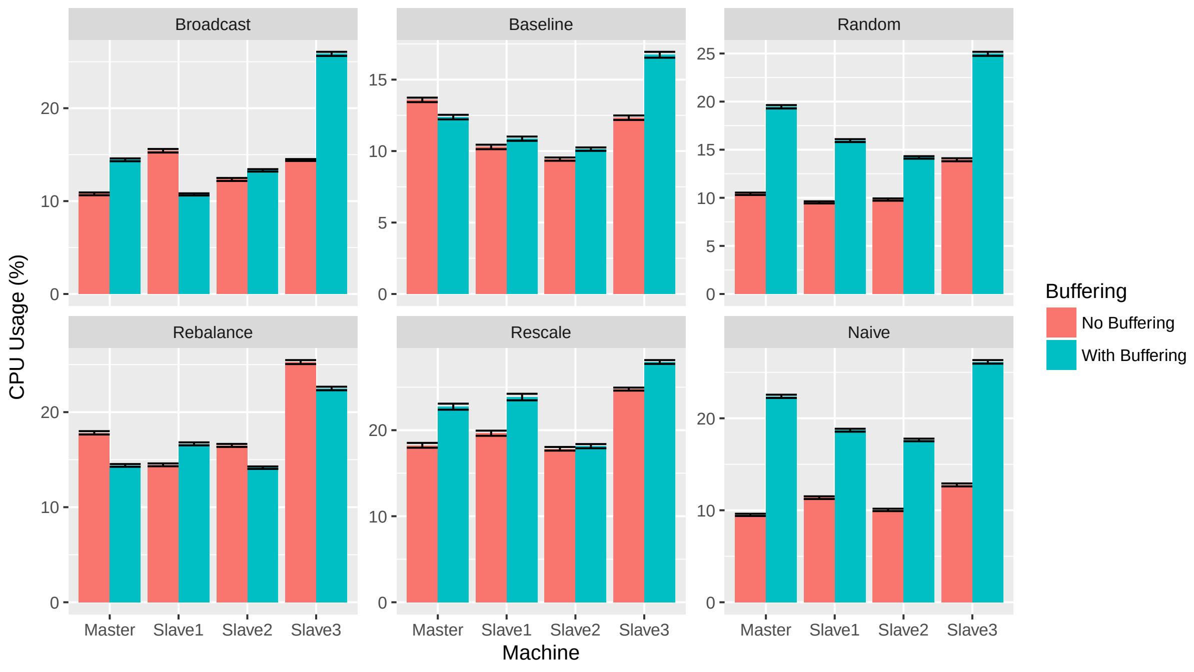
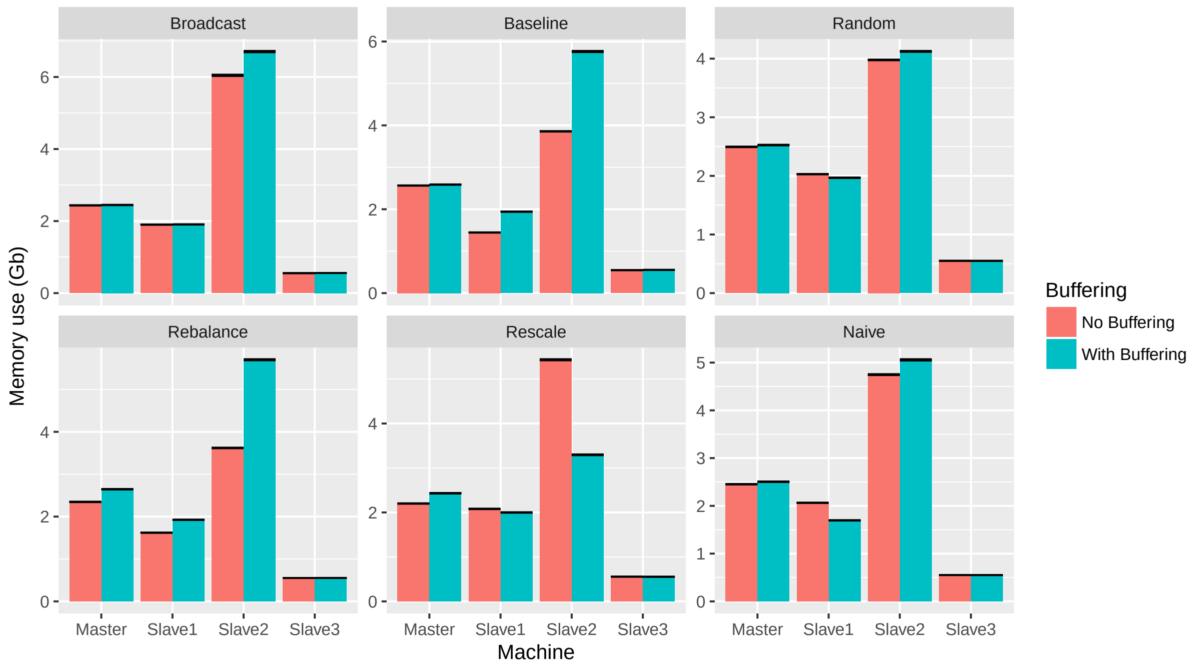


Figure 6.8: Average Memory Usage per node

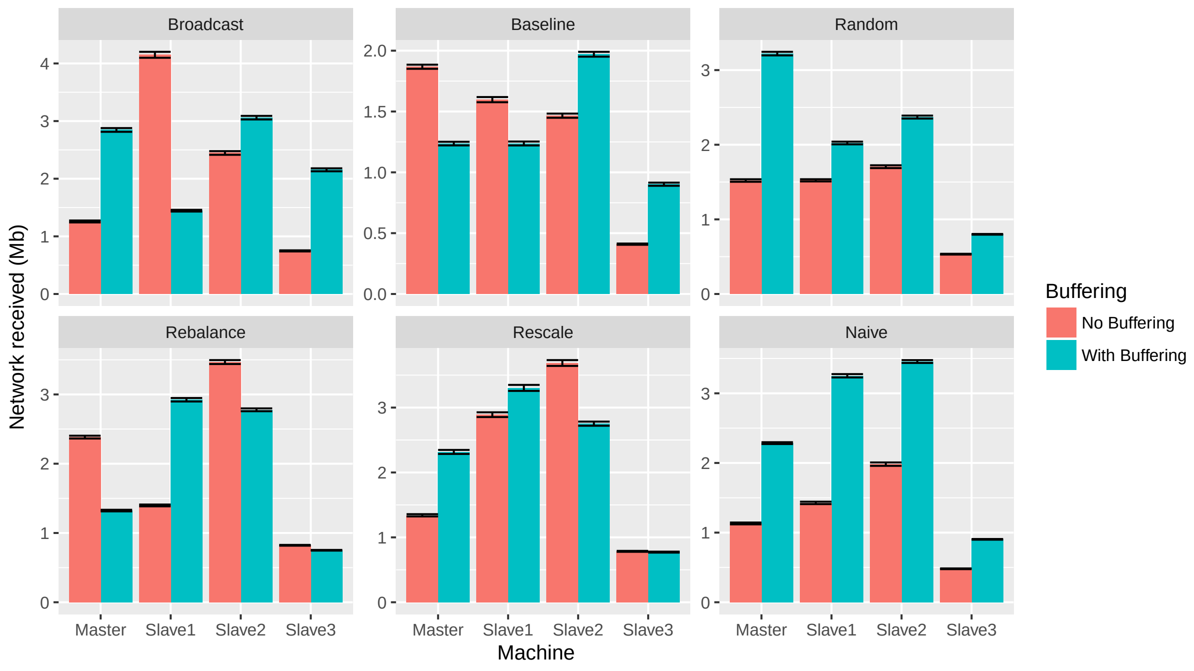


in slave2, and the rescale which is outperformed regarding memory usage in slave two not using the buffering.

In general the memory usage achieve the max available for each machine, just particular cases discussed in the last paragraph could be improved and worsened.

The incoming data from the network, which can be from the partition of the broker

Figure 6.9: Average Network Receive per node



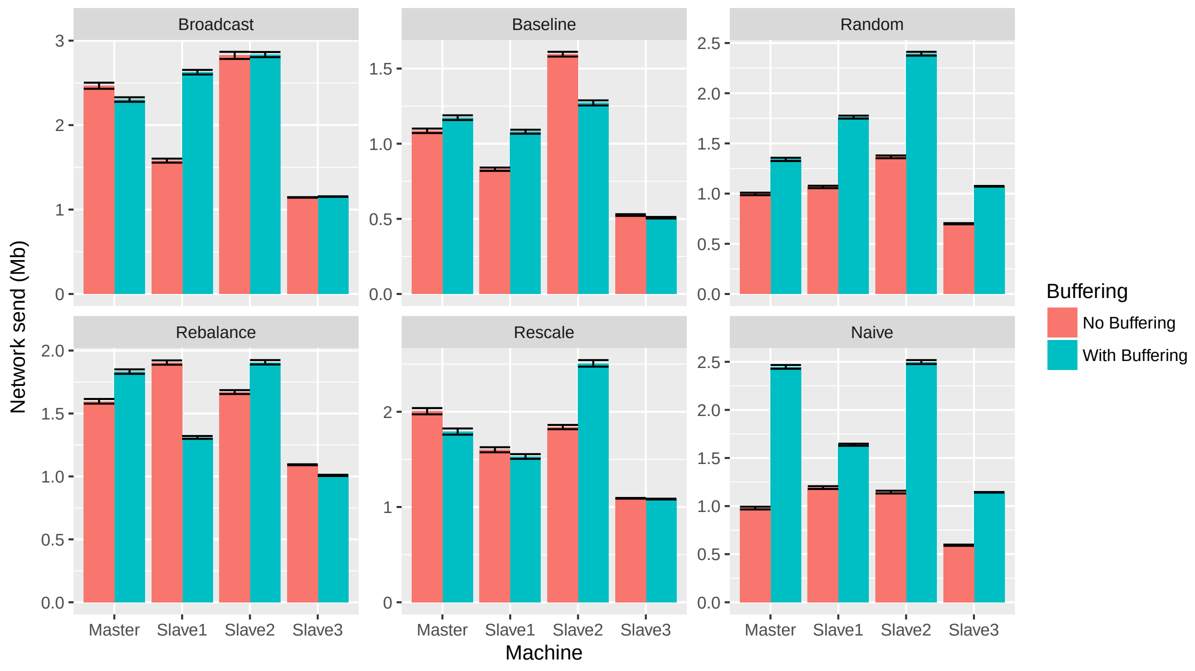
or from the machines inside the cluster, is presented in Figure 6.9. An Overall vision is that the buffering approach provides a significantly higher throughput for every machine. In the rescale algorithm it is possible to see a more comparable data incoming, matching the capability of the machines. That behavior can also be seen in naive, however naive does not achieve such a high throughput as rescale, it can be explained since the rescale distribute the data assuming information that it already has.

The naive requires overhead to read the incoming ratio of each machine to provide the data. It performs better using the buffering since it needs to make a single decision for 128 events, when not buffering it requires a choice for a single event, therefore achieve greater overhead.

Thus, an interesting fact is that rescale, broadcast and baseline present a higher variation of data received, for another side, the others algorithms maintain stable. In the rescale it is mostly found in the slaves, for the broadcast and baseline are found instability for all machines. Besides, the baseline achieves the high incoming ratio in the master node, therefore proving that the partitions are feed most of the master node, requiring more steps of shuffling to reduce the dependency between the nodes.

The network output data of the every data is measured, the output can be from the results that every machine is producing or from the shuffling phase of map reduce, which is the communication between the machines to execute reduces tasks without any dependency of values. It is shown in Figure 6.10, in most of the cases the buffering increases the network

Figure 6.10: Average Network Send per node



out utilization.

The broadcast and rebalance algorithms present a lot of over communication between nodes, since they present the higher output ratio. This happens in the broadcast since the messages are replicated for every machine, thus requiring a lot of effort to broadcast the events. The rebalance seems to present the same case, since it need to communicate with the others machines of the cluster.

The rescale, naive and random seems to centralize its operations inside each machine, thus not requiring to create the over communication. There is a few increase in network usage using the buffering in rescale, and in random and naive seems it increases even more. Principally, since the buffering approach can increase the throughput and the reflex can be seen here.

6.4 Discussion

The use of the Aten dispatcher module provides improvements in either configuration it gives. The evaluation provided covers the different scenarios it can reach, and for most of them, it achieves high throughput and performance, necessary for the real-time constraints. Only the buffering approach already promotes enhancements in the stream processing applications, showing that the real-time requirements can be achieved even

Table 6.3: Average Execution time comparison

	Total Execution Time	Gain (%)
Baseline	10052	0
Broadcast	6384	36.49
Naive	6445	35.88
Random	7258	27.79
Rebalance	8042	19.99
Rescale	2895	71.19

with greater throughput ratio.

The Table 6.3 present the total duration time to consume and process the whole events of the dataset, it is from the average total time of all replications. To calculate the gain it takes into account the baseline time, with the default execution which does not apply any perk from the dispatcher.

For any algorithm that the dispatcher module implements, it is evident the improvements it provides. The Rescales presents up the better result so far, with 71 % gain, followed by the Broadcast and Naive. Even with a bad load distribution of the Rebalance and Random, it was possible to achieve better results than the baseline.

7 FINAL CONSIDERATIONS

In this chapter, the research conclusions are presented, which could be achieved during the development of this work, and the future work which can use this proposal as the base. All the points from this chapter stand on the considerations of the author.

7.1 Conclusions

Aten is a high-level model that can be placed between a message queue and hybrid processing engine, seeking to optimize the communication and data flow to achieve improved data distribution within heterogeneous systems and culminate the usage of network, CPU, and Memory.

Stream processing system, even from just being part of a hybrid processing system, requires several optimizations in order to provide instability and reliability in Big Data applications. Aten proposes two different forms of optimization: the aggregation also called batching, and the management of physical partitions. Both of these solutions depend on each other, meaning that if there is no aggregation approach the physical partitioning will not be so useful and vice-versa. However, it is not simple to use stream optimization, assuming that for every scenario a different best suitable solution will be found. Therefore, Aten provides insight into these optimizations in order to assist the conception of future Big Data solutions.

The definition of an ideal and global aggregation size is a challenge in the stream processing system. Every application has its particular communication messages which comprise different sizes, frequency, and requirements of the deadline. The ideal is to define the aggregation, not by the amount of messages, but by the total size in bytes of the new batch (or message). The size would depend on the round time trip of the events, the bandwidth and the transmission capacity assuming no noise. Although, it would still have some open discussion about this approach since are various variables which variate continuously and do not follow a particular pattern. In this proposal, initial experiments are used to found the convergence point in which the "ideal" size of aggregation, therefore it follows an empirical methodology. However, it is necessary to define a dynamic size, that would adapt to environmental changes and resource fluctuation.

Moreover, the results show the impact of different algorithms in Aten, allowing throughput and resource exploitation to be increased. During the conception of this work, it was possible to found different approaches to partition data. Even though, a particular algorithm was proposed which attempt to achieve the goals proposed in this research. Also, the metrics provided insightful results which allowed to acknowledge from the algorithm of events aggregation and the algorithms of data partitioning inside the proposed model.

The dispatcher module implements the evaluated algorithms and allows the usage of a single one per run. Therefore, all possible combinations of configurations of the dispatcher were considered in its evaluation. The partition algorithm called *Rescale* turns out to be the most effective for every scenario, but it may not be the same for different applications under different environment configurations. The naive algorithm, which is proposed in this research, also performed efficiently comparing most solutions that are present in the state-of-the-art.

It is also noted that even for algorithms, that shown poor performance like *Random* and *Rebalance*, it was possible to increase its throughput by changing the default batch size. However, in order to maximize, even more, the throughput of the streaming systems a more detailed evaluation is required, since the performance is workload, network capacity (i.e., bandwidth, transference capacity, noise) and application sensitive.

The Aten module was designed with the premise to be generic and allow its manipulation to adapt to several scenarios. The model itself enables the configuration, and its components can be replaced or changed in order to achieve its adaptability.

7.2 Future Work

In future works, it is possible to evaluate dynamic approaches that self-adapt to changes such as data skew and volatile environments in the multi-cloud environment. These approaches concern the size of the buffer assuming application constraints (i.e., the window size), and provide dynamic and adaptive algorithms to partition data stream over environmental changes with further experiments. Moreover, the definition of a dynamic aggregation size, batching length or breadth is still an open problem which presents an opportunity that could be explored in future works.

The Naive algorithm which was proposed in this proposal also has room for enhancements. For instance, it is possible to consider not only the current throughput in the application, but may consider the resources availability and capacity. Assuming that the Rescale algorithms analysis the parallelism capacity of the operators, if the naive algorithms also find the resource CPU frequency, parallelism and cache it would be possible to achieve perhaps more promising results.

Besides, there is plenty of space for different proposals of data distribution, either by manipulating the data partitions or applying different stream processing optimization. Therefore, new algorithms can be implemented and proposed to be evaluated in multiple scenarios. Furthermore, it is possible to assess particular techniques using benchmark kits or establishing numerous situations designs using frameworks and applications.

REFERENCES

ABADI, D. J. et al. The design of the borealis stream processing engine. In: **In CIDR**. [S.l.: s.n.], 2005. p. 277–289.

ABAWAJY, J. Comprehensive analysis of big data variety landscape. **Journal of Parallel, Emergent and Distributed Systems. Taylor & Francis**, v. 30, n. 1, p. 5-14, 2015.

AKIDAU, T. et al. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. **Proceedings of the VLDB Endowment**, v. 8, p. 1792–1803, 2015.

ALEXANDROV, A. et al. The stratosphere platform for big data analytics. **The International Journal on Very Large Data Bases**, v. 23, n. 6, p. 939-964, 2014.

AMAZON. **O que são dados em streaming?** 2016. Accessed on: 04/05/2017. Available from Internet: <https://aws.amazon.com/pt/streaming-data/>.

ANJOS, J. C. S. dos et al. Smart: An application framework for real time big data analysis on heterogeneous cloud environments. In: **Proceedings of the International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing, CIT/IUCC/DASC/PICOM'15**. [S.l.]: IEEE, 2015. p. 199–206.

ANJOS, J. C. S. dos et al. Fast-sec: an approach to secure big data processing in the cloud. **International Journal of Parallel, Emergent and Distributed Systems**, v. 0, n. 0, p. 1-16, 2017.

AO, M. D. de A.; VEITH, A. da S.; BUYYA, R. Distributed data stream processing and edge computing: A survey on resource elasticity and future directions. **Journal of Network and Computer Applications**, v. 103, p. 1- 17, 2018.

Apache Spark. **Spark Streaming Programming Guide**. 2014. Accessed on: 04/05/2017. Available from Internet: <http://spark.apache.org/docs/latest/streaming-programming-guide.html>.

ASSUNÇÃO, M. D. et al. Big data computing and clouds: Trends and future directions. **Journal of Parallel and Distributed Computing**, v. 79-80, p. 3 – 15, 2015. ISSN 0743-7315. Special Issue on Scalable Systems for Big Data Management and Analytics. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0743731514001452>.

ATASHKAR, A. H.; GHADIRI, N.; JOODAKI, M. Linked data partitioning for rdf processing on apache spark. In: **2017 3th International Conference on Web Research (ICWR)**. [S.l.: s.n.], 2017. p. 73–77.

BARGA, R. S.; CAITUIRO-MONGE, H. Event correlation and pattern detection in cedr. In: **EDBT Workshops**. [S.l.: s.n.], 2006.

BORTHAKUR, D. **The hadoop distributed file system: Architecture and design**. 2007. Accessed on: 04/05/2017. Available from Internet: <https://svn.apache.org/repos/asf/hadoop/common/tags/release-0.16.0/docs/hdfs_design.pdf>.

BUDDHIKA, T.; PALLICKARA, S. Neptune: Real time stream processing for internet of things and sensing environments. In: **2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)**. [S.l.: s.n.], 2016. p. 1143–1152. ISSN 1530-2075.

BUDDHIKA, T. et al. Online scheduling and interference alleviation for low-latency, high-throughput processing of data streams. **Transactions on Parallel and Distributed Systems. IEEE**, PP, n. 99, p. 1-1, 2017.

CARBONE, P. et al. Apache flink: Stream and batch processing in a single engine. **Bulletin of the IEEE Computer Society Technical Committee on Data Engineering. IEEE**, v. 36, n. 4, p. 17-29, 2015.

CARDELLINI, V.; NARDELLI, M.; LUZI, D. Elastic stateful stream processing in storm. In: **2016 International Conference on High Performance Computing Simulation (HPCS)**. [S.l.: s.n.], 2016. p. 583–590.

CHANDARANA, P. Big Data Analytics Frameworks. **Proceedings of the International Conference on Circuits, Systems, Communication and Information Technology Applications, CSCITA'14**, p. 430–434, April 2014.

CHANDY, K. M.; LAMPORT, L. Distributed snapshots: Determining global states of distributed systems. **ACM Trans. Comput. Syst.**, ACM, New York, NY, USA, v. 3, n. 1, p. 63–75, feb. 1985. ISSN 0734-2071. Available from Internet: <http://doi.acm.org/10.1145/214451.214456>.

DAS, T. et al. Adaptive stream processing using dynamic batch sizing. In: **Proceedings of the ACM Symposium on Cloud Computing**. New York, NY, USA: ACM, 2014. (SOCC '14), p. 16:1–16:13. ISBN 978-1-4503-3252-1. Available from Internet: <http://doi.acm.org/10.1145/2670979.2670995>.

DEAN, J.; GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. **Journal of Communications. ACM**, v. 51, n. 1, p. 107-113, 2008.

DEAN, J.; GHEMAWAT, S. **MapReduce: Simplified Data Processing on Large Clusters**. 2009. <https://research.google.com/archive/mapreduce-osdi04-slides/index.html>. Accessed: 2018-06-30.

DING, M. et al. More convenient more overhead: The performance evaluation of hadoop streaming. In: **Proceedings of the 2011 ACM Symposium on Research in Applied Computation**. New York, NY, USA: ACM, 2011. (RACS '11), p. 307–313. ISBN 978-1-4503-1087-1. Available from Internet: <http://doi.acm.org/10.1145/2103380.2103444>.

D'SOUZA, S.; HOFFMAN, S. **Apache Flume: Distributed Log Collection for Hadoop**. Packt Publishing, Limited, 2013. (Community experience distilled). ISBN 9781782167914. Available from Internet: <https://books.google.com.br/books?id=y0jQnQEACAAJ>.

EKANAYAKE, J. et al. Twister: A runtime for iterative mapreduce. In: **Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing**. New York, NY, USA: ACM, 2010. (HPDC '10), p. 810–818. ISBN 978-1-60558-942-8. Available from Internet: <http://doi.acm.org/10.1145/1851476.1851593>.

FERNANDEZ, R. C. et al. Liquid: Unifying nearline and offline big data integration. In: **CIDR**. [S.l.: s.n.], 2015.

GAD, S. **When to Sell Stocks**. 2017. Accessed on: 07/08/2017. Available from Internet: <Retrievedfrom:http://www.investopedia.com/articles/stocks/10/when-to-sell-stocks.asp>.

GEDIK, B. Partitioning functions for stateful data parallelism in stream processing. **The VLDB Journal**, v. 23, n. 4, 2014.

GHEMAWAT, J. D. S. **MapReduce: Simplified Data Processing on Large Clusters**. 2004. Accessed on: 07/08/2017. Available from Internet: <Retrievedfrom:http://labs.google.com/papers/mapreduce.html>.

GOYAL, A.; BHARTI. Study on emerging implementations of mapreduce. In: **International Conference on Computing, Communication Automation**. [S.l.: s.n.], 2015. p. 16–21.

GRADVOHL, A. L. S. et al. Comparing distributed online stream processing systems considering fault tolerance issues. **Journal of Emerging Technologies in Web Intelligence**, v. 6, n. 2, p. 174-179, 2014.

HADOOP, A. **Hadoop**. 2009. Accessed on: 04/05/2017. Available from Internet: <http://hadoop.apache.org>.

HASANI, Z.; KON-POPOVSKA, M.; VELINOV, G. Lambda architecture for real time big data analytic. In: . [S.l.: s.n.], 2014. p. 133–143.

HASHEM, I. A. T. et al. The rise of “big data” on cloud computing: Review and open research issues. **Journal of Information Systems. Elsevier**, v. 47, n. Supplement C, p. 98-115, 2015.

HILBERT, M. Big data for development: A review of promises and challenges. **Journal of Development Policy Review**, v. 34, n. 1, p. 135-174, 2016.

HIRZEL, M. et al. A catalog of stream processing optimizations. **Journal of Computing Surveys. ACM**, v. 46, n. 4, p. 46:1-46:34, 2014.

HIVE, A. **Apache Hive**. 2015. Accessed on: 04/05/2017. Available from Internet: <https://hive.apache.org/e>.

HUANG, B. et al. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In: **Proceedings of the 26th International Conference on Data Engineering Workshops. ICDEW’10**. [S.l.]: IEEE, 2010. p. 41–51.

INC, A. **Apache Spark: Lightning-fast cluster computing**. 2016. Accessed on: 07/08/2017. Available from Internet: <Retrievedfrom:http://spark.apache.org>.

IONESCU, V. M. The analysis of the performance of rabbitmq and activemq. In: **2015 14th RoEduNet International Conference - Networking in Education and Research (RoEduNet NER)**. [S.l.: s.n.], 2015. p. 132–137. ISSN 2068-1038.

JAIN, R. **The art of computer systems performance analysis - techniques for experimental design, measurement, simulation, and modeling**. 2rd. ed. [S.l.]: Wiley, 1991. 720 p. ISBN 978-0-471-50336-1.

JAVED, M. H.; LU, X.; PANDA, D. K. D. Characterization of big data stream processing pipeline: A case study using flink and kafka. In: **Proceedings of the Fourth IEEE/ACM International Conference on Big Data Computing, Applications and Technologies**. New York, NY, USA: ACM, 2017. (BDCAT '17), p. 1–10. ISBN 978-1-4503-5549-0. Available from Internet: <http://doi.acm.org/10.1145/3148055.3148068>.

JONES, B. et al. Rabbitmq performance and scalability analysis. **project on CS**, v. 4284, 2011.

JUNIOR, P. R. R. de S. et al. Aten: A Dispatcher for Big Data Applications in Heterogeneous Systems. In: . [S.l.]: IEEE Computer Society, 2018. (HPCS - International Conference on High Performance Computing and Simulation), p. 585–592. ISBN 978-1-5386-7879-4.

KAPPA Architecture - Questioning the Lambda Architecture. 2014. <https://www.oreilly.com/ideas/questioning-the-lambda-architecture>. Accessed: 2018-06-30.

KARGER, D. et al. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In: **Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing**. New York, NY, USA: ACM, 1997. (STOC '97), p. 654–663. ISBN 0-89791-888-6. Available from Internet: <http://doi.acm.org/10.1145/258533.258660>.

KATSIFODIMOS, A.; SCHELTER, S. Apache flink: Stream analytics at scale. In: **2016 IEEE International Conference on Cloud Engineering Workshop (IC2EW)**. [S.l.: s.n.], 2016. p. 193–193.

KIRAN, M. et al. Lambda architecture for cost-effective batch and speed big data processing. In: **2015 IEEE International Conference on Big Data (Big Data)**. [S.l.: s.n.], 2015. p. 2785–2792.

KLEPPMANN, M.; KREPS, J. Kafka, samza and the unix philosophy of distributed data. **IEEE Data Eng. Bull.**, v. 38, n. 4, p. 4–14, 2015.

KODALI, R. K.; SORATKAL, S. Mqtt based home automation system using esp8266. In: IEEE. **Humanitarian Technology Conference (R10-HTC), 2016 IEEE Region 10**. [S.l.], 2016. p. 1–5.

KREPS, J. Kafka : a distributed messaging system for log processing. In: . [S.l.: s.n.], 2011.

KREPS, J.; NARKHEDE, N.; RAO, J. Kafka: A distributed messaging system for log processing. In: **Proceedings of 6th International Workshop on Networking Meets Databases (NetDB), Athens, Greece**. [S.l.: s.n.], 2011.

LAKSHMAN, A.; MALIK, P. Cassandra: A decentralized structured storage system. **SIGOPS Oper. Syst. Rev.**, ACM, New York, NY, USA, v. 44, n. 2, p. 35–40, abr. 2010. ISSN 0163-5980. Available from Internet: <http://doi.acm.org/10.1145/1773912.1773922>.

LI, J. et al. Enabling elastic stream processing in shared clusters. In: **2016 IEEE 9th International Conference on Cloud Computing (CLOUD)**. [S.l.: s.n.], 2016. p. 108–115.

LIGHT, R. A. Mosquitto: server and client implementation of the mqtt protocol. **Journal of Open Source Software**, The Open Journal, v. 2, n. 13, 2017.

LIN, J. The lambda and the kappa. **IEEE Internet Computing**, v. 21, n. 5, p. 60–66, 2017. ISSN 1089-7801.

MARZ, N.; WARREN, J. **Big Data: Principles and Best Practices of Scalable Realtime Data Systems**. 1st. ed. [S.l.]: Manning Publications Co., 2015. 328 p. ISBN 1617290343, 9781617290343.

MATTEUSSI, K. Um Estudo Sobre a ContenÇão de Disco em Ambientes Virtualizados Utilizando Contêineres e Seu Impacto Sobre Aplicações MapReduce. p. 94, 2016. Dissertação de Mestrado, Programa de Pós-Graduação em Ciência da Computação, PUCRS. 2016. pp.

MATTEUSSI, K.; ROSE, C. D. Uma estratégia de alocaç ao dinâmica e preditiva de recursos de I/O para reduzir o tempo de execuç ao em aplicaç oes mapreduce. In: **Escola Regional de Alto Desempenho do Estado do Rio Grande do Sul**. [S.l.: s.n.], 2015. p. 45–47. ISBN 85-88442-74-4.

MATTEUSSI, K. J. et al. Um guia para a modelagem e desenvolvimento de aplicações big data sobre o modelo de arquitetura zeta. In: **Escola Regional de Alto Desempenho do Estado do Rio Grande do Sul**. [S.l.: s.n.], 2017. ISBN 85-88442-74-4.

MILOSLAVSKAYA, N.; TOLSTOY, A. Application of big data, fast data, and data lake concepts to information security issues. In: **Proceedings of the 4th International Conference on Future Internet of Things and Cloud Workshops. FiCloudW'16**. [S.l.]: IEEE, 2016. p. 148-153.

MORIK, K. A survey of the stream processing landscape. In: . [S.l.: s.n.], 2014.

NASIR, M. A. U. et al. Load balancing for skewed streams on heterogeneous cluster. **CoRR**, abs/1705.09073, 2017.

NASIR, M. A. U. et al. Partial key grouping: Load-balanced partitioning of distributed streams. **CoRR**, abs/1510.07623, 2015.

NEUMEYER, L. et al. S4: Distributed stream computing platform. In: **Data Mining Workshops (ICDMW), 2010 IEEE International Conference on**. [S.l.: s.n.], 2010. p. 170–177.

NOYES, D. **The Top 20 Valuable Facebook Statistics**. 2015. Accessed on: 04/05/2017. Available from Internet: <Retrievedfrom:https://zephoria.com/social-media/top-15-valuable-facebookstatistics>.

PALLICKARA, S.; EKANAYAKE, J.; FOX, G. Granules: A lightweight, streaming runtime for cloud computing with support, for map-reduce. In: **2009 IEEE International Conference on Cluster Computing and Workshops**. [S.l.: s.n.], 2009. p. 1–10. ISSN 1552-5244.

PATHAK, H.; RATHI, M.; PAREKH, A. Introduction to real-time processing in apache apex. In: **National Conference "NCPCI**. [S.l.: s.n.], 2016. v. 2016, p. 19.

PHAM, L. M. et al. Cirus: an elastic cloud-based framework for ubilytics. **Annals of Telecommunications**, v. 71, n. 3, p. 133-140, 2016.

PHAM, L. M. et al. Roboconf: A hybrid cloud orchestrator to deploy complex applications. In: **2015 IEEE 8th International Conference on Cloud Computing**. [S.l.: s.n.], 2015. p. 365–372. ISSN 2159-6182.

PIG, A. **Apache Pig**. 2015. Accessed on: 04/05/2017. Available from Internet: <https://pig.apache.org/>.

POLATO, I. et al. A Comprehensive View of Hadoop Research—A Systematic Literature Review. **Journal of Network and Computer Applications**, v. 46, p. 1–25, 2014. ISSN 1084-8045. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S1084804514001635>.

RENART, E. G.; DIAZ-MONTES, J.; PARASHAR, M. Data-driven stream processing at the edge. In: **Proceedings of the 1st International Conference on Fog and Edge Computing. ICFEC'17**. [S.l.]: IEEE, 2017. p. 31–40.

RISTA, C. et al. Improving the network performance of a container-based cloud environment for hadoop systems. In: **Proceedings of the International Conference on High Performance Computing Simulation. HPCS'17**. [S.l.]: IEEE, 2017. p. 619–626.

SILVA, V. A. da et al. Strategies for big data analytics through lambda architectures in volatile environments. In: **Proceedings of Preprint arXiv. TA'2017**. [S.l.]: Elsevier, 2016. p. 114-119.

SNYDER, B.; BOSANAC, D.; DAVIES, R. Introduction to apache activemq. **Active MQ in Action**, p. 6–16, 2017.

STONEBRAKER, M.; ROWE, L. A. The design of postgres. **SIGMOD Rec.**, ACM, New York, NY, USA, v. 15, n. 2, p. 340–355, jun. 1986. ISSN 0163-5808. Available from Internet: <http://doi.acm.org/10.1145/16856.16888>.

STORM, A. **Storm Documentation**. 2014. Accessed on: 04/05/2017. Available from Internet: <http://storm.apache.org/documentation/Home.html>.

THEIN, K. Apache kafka: Next generation distributed messaging system. **International Journal of Scientific Engineering and Technology Research**, v. 3, n. 47, p. 9478–9483, 2014.

TOSHNIWAL, A. et al. Storm at twitter. In: **Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data**. New York, NY, USA: ACM, 2014. (SIGMOD '14), p. 147–156. ISBN 978-1-4503-2376-5. Available from Internet: <http://doi.acm.org/10.1145/2588555.2595641>.

TUDORAN, R.; COSTAN, A.; ANTONIU, G. Overflow: Multi-site aware big data management for scientific workflows on clouds. **IEEE Transactions on Cloud Computing**, v. 4, n. 1, p. 76–89, Jan 2016. ISSN 2168-7161.

TUDORAN, R. et al. Jetstream: Enabling high throughput live event streaming on multi-site clouds. **Journal of Future Generation Computer Systems. Elsevier**, v. 54, n. Supplement C, p. 274-291, 2016.

TUDORAN, R. et al. Evaluating streaming strategies for event processing across infrastructure clouds. In: **Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on**. [S.l.: s.n.], 2014. p. 151–159.

TUDORAN, R. et al. Evaluating streaming strategies for event processing across infrastructure clouds. In: **Proceedings of the 14th International Symposium on Cluster, Cloud and Grid Computing. CCGrid'14**. [S.l.]: IEEE/ACM, 2014. p. 151-159.

TUDORAN, R. et al. Jetstream: Enabling high performance event streaming across cloud data-centers. In: **Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems**. New York, NY, USA: ACM, 2014. (DEBS '14), p. 23–34. ISBN 978-1-4503-2737-4. Available from Internet: <http://doi.acm.org/10.1145/2611286.2611298>.

VAVILAPALLI, V. K. et al. Apache hadoop yarn: Yet another resource negotiator. In: **Proceedings of the 4th annual Symposium on Cloud Computing, SOCC'13**. [S.l.]: ACM, 2013. p. 5.

VERNER, U.; SCHUSTER, A.; SILBERSTEIN, M. Processing data streams with hard real-time constraints on heterogeneous systems. In: **Proceedings of the international conference on Supercomputing**. [S.l.]: ACM, 2011. p. 120-129.

WELSH, M.; CULLER, D.; BREWER, E. Seda: An architecture for well-conditioned, scalable internet services. **Journal of Operating Systems Review. ACM**, v. 35, n. 5, p. 230-243, 2001.

WHITE, T. **Hadoop: The Definitive Guide**. 1st. ed. [S.l.]: O'Reilly Media, Inc., 2009. 688 p. ISBN 978-1-449-31152-0.

WHITE, T. **Hadoop: The Definitive Guide**. [S.l.]: O'Reilly Media, Inc., 2012. ISBN 1449311520, 9781449311520.

XAVIER, M. G. et al. Understanding performance interference in multi-tenant cloud databases and web applications. In: **Proceedings of the International Conference on Big Data, Big Data'16**. [S.l.]: IEEE, 2016. p. 2847–2852.

XAVIER, M. G. et al. A performance isolation analysis of disk-intensive workloads on container-based clouds. In: **Proceedings of the 23rd Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP'15**. [S.l.]: IEEE, 2015. p. 253-260.

XHAFA, F.; NARANJO, V.; CABALLé, S. Processing and analytics of big data streams with yahoo!s4. In: **2015 IEEE 29th International Conference on Advanced Information Networking and Applications**. [S.l.: s.n.], 2015. p. 263–270. ISSN 1550-445X.

ZAHARIA, M. et al. Fast and interactive analytics over hadoop data with spark. **Usenix Login**, v. 37, n. 4, p. 45–51, 2012.

ZAHARIA, M. et al. Discretized streams: Fault-tolerant streaming computation at scale. In: **Proceedings of the 24th Symposium on Operating Systems Principles, SOSP'13**. [S.l.]: ACM, 2013. p. 423-438.

ZAHARIA, M. et al. Apache spark: A unified engine for big data processing. **Journal of Communications. ACM**, v. 59, n. 11, p. 56-65, 2016.

ZHUANG, Z. et al. Effective multi-stream joining in apache samza framework. In: **Proceedings of the International Congress on Big Data. BigData Congress'16**. [S.l.]: IEEE, 2016. p. 267-274.

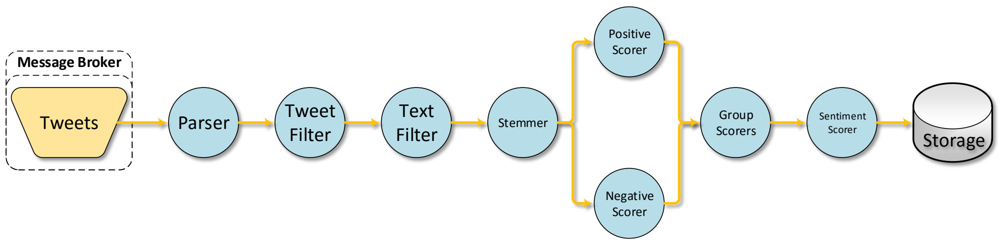
APPENDIX A — BUFFERING INITIAL EXPERIMENTS

The present appended chapter describes the environment setup and results of a primary evaluation to demonstrate the impacts of the buffering inside the Aten model. The experiments comprise empirical evaluation performed on the Microsoft Azure cloud, at the PaaS level, using a VM on West of US as the Data Source, a VM on North of EU as the Message Broker and three VMs on East of Japan as the Stream-processing set. The system runs on A10 instances (processor Intel® Xeon® E5-2670 @ 2.6 GHz, 8 cores, 56GB DDR3-1600 MHz RAM and 120 GB storage) with Ubuntu Server. Each VM instance synchronizes its time with the NTP Server. Experiments were designed based on the methodology proposed by (JAIN, 1991) with a replication factor equal to 20 and 7 batch sizes. The experiments were performed as the following tools:

- **Data Source**: A Java implementation to produce parallel events;
- **Message Broker**: Apache Kafka 0.10.0;
- **Stream-processing Engine**: Apache Flink.

The stream-processing models evaluated were **One-at-a-time**: one event at a time and the **Batch-based**: buffer sizes of 2, 10, 150, 250, 500 and 1000 events. In order to evaluate the proposed environments the following tools were used: Iperf[1] to measure the throughput and the bandwidth among the evaluated regions; Dstat-monitor[2] was used for measuring resource consumption, CPU utilization, memory usage, disk and network I/O; Hping[3] was used to obtain information regarding the network latency between the test environments; Ganglia[4] has been used to monitor the reception and sending messages time through Apache Kafka and Flink.

Figure A.1: Stream Processing Flow. Designed by the author



The evaluated application is a sentimental analysis application Figure A.1 as Stream

[1]https://iperf.fr
[2]http://dag.wiee.rs/home-made/dstat/
[3]http://www.hping.org/
[4]http://ganglia.sourceforge.net/

Processing Motor, where tweets were classified. The application derivates from a Natural Processing Language algorithm, based on a dictionary. First, it applies a parse in the twitter text information, so it can pass throw a filter that removes the tweet from different languages and remove propositional words. Then, the information goes to a stemmer and a scorer, to classify the tweet based and two classes: active and adverse. Finally, the results are stored.

In order to evaluate the proposed environments the following tools were used: Iperf to measure the throughput and the bandwidth among the evaluated regions; Dstat-monitor was used for measuring resource consumption, CPU utilization, memory usage, disk and network I/O; Hping was used to obtain information regarding the network latency between the test environments; Ganglia has been used to monitor the reception and sending messages time through Kafka.

Tests were performed in 140 runs, where the size of the tweet was $\approx 2KB$. In each run were sent 1.000,000 tweets where the experiment time took $\approx 12\ hours$.

A.0.1 Results

In this subsection, we present the results. The Figure A.2 shows the entire execution over the decoupled environment, shows the complete execution over all proposed scenarios, i.e., batch sizes. We noticed a significant variation in sending smaller batches. Most of it, caused by the substantial number of messages sent over the network, being more susceptible to failures and high latency that is caused mainly by the cloud services or the network links.

Although, bigger batches performed less variation and lower execution time. We can observe that in bigger batches there is fewer event consumption but with smaller processing time. Showing that for that type of environment is viable to send bigger messages than send more amounts of smaller messages. Batch sizes of 250 to 1000 shown promising results, providing stability and a shorter run time.

Figure A.2: Execution time over the entire model

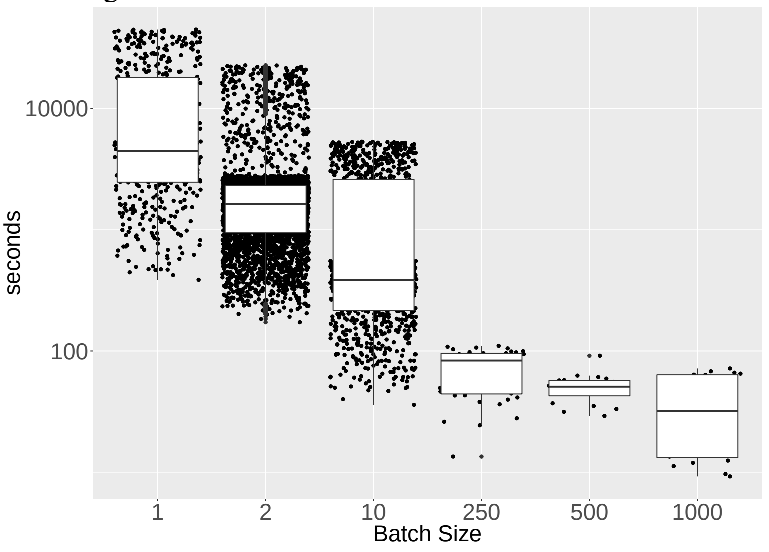
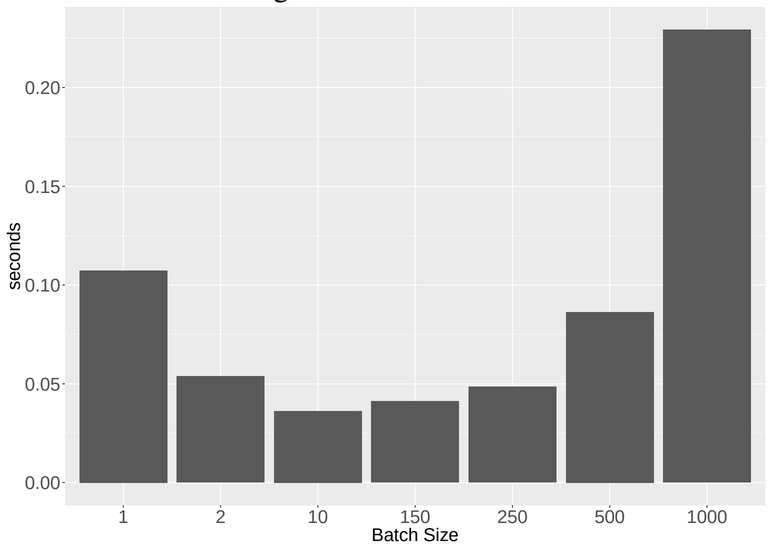


Figure A.3: Event time

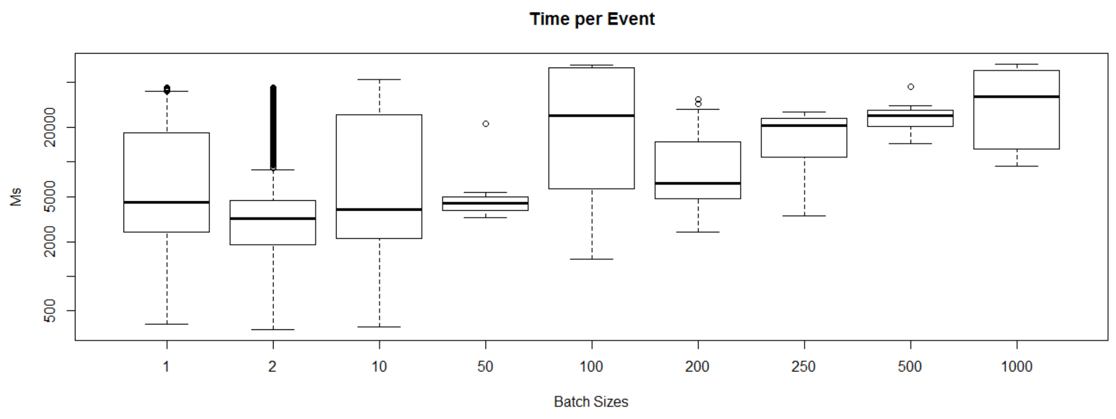


Obviously, we also need to consider as well the characteristics of the evaluated application, that may benefit the approach. For this reason, the chosen application is I/O bound. It is because most of the Big Data Stream applications are considered I/O bound (HUANG et al., 2010) the strategy may benefit most of the applications.

Moreover, it is possible to conclude that the aggregation is feasible since there is no further increase in execution time inside Flink. It also may prove, that there is an improvement of exploitation from the available resources since the throughput increased, but it needs further and deeply evaluations.

As seen in Figure A.2 the batch sizes of 1 to 10 are significantly inferior in execution time, since the involved logistics to send plenty of messages besides using a bigger batch with multiples streams. At this point, Figure A.3 reinforces that idea, considering that it presents the entire throughput time to prepare, send and process a batch-sized.

Figure A.4: Time by event



In our results we noticed a great variation in sending small batches as can be seen in Figure A.4, because of the largest number of messages sent over the network, being more susceptible to failures. For the biggest batches there is less variation and the event duration time is smaller showing that for that type of network is more viable to send bigger batches than send more amounts of smaller messages.

APPENDIX B — CAPÍTULO DE RESUMO EM PORTUGUÊS

B.1 Introdução

Diversas organizações e companhias extraem informações do grande volume de dados (i.e., Big Data) gerados por diversos dispositivos e consumidores. Tais informações geralmente tem como objetivo melhorar a tomada de decisões, porém requerem uma grande capacidade de recursos computacionais, visto a complexidade do processo de análise de dados.

A introdução do modelo MapReduce permitiu a análise destes grandes volumes de dados e com seu vasto uso foi subdividido em duas subclasses: Batch Processing e Stream Processing. No Batch Processing os dados são manipulados/analisados sem necessitar de uma resposta em tempo real. Já no Stream Processing, são processadas pequenas parcelas de dados, também chamados eventos, na casa de kilobytes.

A computação em Cloud fornece ambientes que beneficiam o processamento Big Data, já que possui um extenso número de datacenters distribuídos geograficamente, tornando mais simples a implementação de aplicações distribuídas. Além de serem facilmente escaláveis tanto verticalmente quanto horizontalmente, assim provendo um ambiente propício para assegurar as necessidades da análise de dados. Apesar disso, é comum que a Cloud seja constituída de dispositivos heterogêneos e com latências de redes diferentes, esses ambientes são conhecidos como multi-clouds (MC). Estas diferenças acarretam em problemas que precisam de atenção como mensagens perdidas ou com atraso, e até falta ou uso indevido dos recursos computacionais causados pelo desbalanço de carga.

Para evitar os problemas acima descritos, este trabalho propõe o Aten, um modelo de *dispatcher*, que visa gerenciar os dados de Batch e Stream processing, aplicando técnicas de otimização na comunicação e no balanço de carga. Além disso, existem como objetivos, a dimnuição da latência na trasmissão de mensagens, utilizar algoritmos de particionamento evitando a troca e dependência de dados entre os nós de processamento e o aumentar o *throughput* de informações.

B.2 Background em Big Data

As aplicações Big Data estão cada vez mais populares sendo adotadas em diversas áreas da sociedade sejam elas sociais ou ecônomicas. Como exemplo, no mercado financeiro os investidores fazem diversas transações de compra e venda conforme os valores sobem e descem. Logo, utilizar uma aplicações que calculam tendências de mercado ajudaria na tomada de decisões, além de proporcionar maior velocidade nas transações.

Na literatura, Big Data é caracterizado pelos aspectos conhecidos com 5 V's, sendo três deles com maior importância para área acadêmica e os outros para a área industrial.

- **Volume**: Relacionado à magnitude, ou seja, está em constante expansão;
- **Velocidade**: Estes dados são gerados rapidamente exigindo ser tratadas com mesma velocidade;
- **Variedade**: Os dados são produzidos em diferentes contextos, sendo estruturados como logs de e-commerce ou não estruturados na forma de video, imagens e etc;
- **Valor**: Esses dados de forma bruta podem não ter significado, porém quando agregados, processados e analisados deve gerar informações estratégicas relevantes para *Business Inteligence*;
- **Veracidade**: Trata-se da confiabilidade da origem dos dados, já que a discretização dos fatos podem gerar incertezas.

Conforme dito, esses dados sempre estão aumentando, possuem os mais diversos formatos e tem alta aplicação no mundo real. Considerando essas características, manipular e analisar esse grande volume de dados de forma rápida e barata se tornou um grande desafio. Com o objetivo de ultrapassar estes desafios, foi introduzido pelo Google, o modelo MapReduce. Separado em duas partes, o mapeameto de uma única instância dos dados como entrada, entre os nós de processamento, e produzir um conjunto chave-valor intermediário e depois agrupar esses conjuntos pelas chaves. Então, durante a etapa de *reduce*, toma como entrada uma chave e todos os valores gerados pelo *map* para aquela determinada chave. Por fim, esses valores são agregados ou combinados para gerar um conjunto menor de dados também representados como pares chave-valor.

Existem duas grandes classes de processamento Big Data, Batch e Stream. O processamento em Batch consiste em analisar grandes volumes de dados em pedaços, chamados *chunks*, geralmente de 32 MB até 64 MB, mas que também pode ser definido pelo programador. O Apache Hadoop foi a primeira ferramenta a utilizar o paradigma MapReduce para processar *Batches*. Para o processamento de grandes arquivos, *Batch processing* depende da replicação no sistema de arquivos, o Hadoop usa o HDFS (the Hadoop Distributed File System), extremamente tolerante a falhas, com baixo custo de hardware e podendo ser utilizado em diversas aplicações, essas diferenças foram cruciais para reproduzir o MapReduce em ambientes distribuídos.

Nos ambientes propícios para Stream processing, os dados são gerados continuamente, simultaneamente e com velocidades diferentes. Geralmenta são pequenos (na casa dos kilobytes) e tem formatos variados devido a grande variedade de dispositivos que podem gerar tais dados, como sensores, videogames, smartphones e etc. As aplicações Stream podem ser representadas por grafos direcionados, onde os vértices são os operadores e as arestas são o fluxo de dados. Os operadores podem ser utilizados para stream ingestion, computation ou data sink. Assim como as aplicações o conceito geral de Stream é modelado através de grafos como visto na Figura 2.3 onde possui três elementos:

- **Fonte de dados**: Produz ou coleta dados;
- **Nó de processamento**: Processa os elementos emitidos pela fonte de dados;
- **Data Sink**: Armazena as informações geradas pelo fluxo de dados.

Entretanto, o processamento Big Data necessita se um alto poder computacional para atender a velocidade e o valor das informações geradas, porém o custo de infraestrutura aumenta significativamente já que a soluções não são escalam proporcionalmente. A partir disto, os ambientes de computação em nuvem se tornoram populares por possuir diversas características para aplicações Big Data, como segurança, flexibilidade, *pay-per-use* e armazenamento escalável. Apesar disso, necessitou-se de modelos que otimizassem a utilização dos recursos da nuvem, entre eles estão a arquitetura Lambda e Kappa.

A arquitetura Lambda, proposta por Marz, é composta por três camadas: Batch, RT e Service. A camada de Batch serve como uma cópia dos dados e pré-computa eles. Já a camada RT processa pequenos datasets de acordo com uma janela de tempo (60 segundos, por exemplo), permitindo a análise em tempo real, e na camada de serviço os resultados são combinados permitindo a interação do usuário com as análises. Em geral, a arquitetura Kappa é análoga à arquitetura Lamda, porém no Kafka Cluster os dados são pré-processados e tudo é tratado como fluxo de dados, ou seja, o foco são eventos em RT, apesar disso não há grande diferença de desempenho em comparação à Lambda.

B.3 Trabalhos Relacionados

A orquestracção de tarefas é crucial em sistemas de processamento Big Data, muitas vezes fornecido pelos próprios *frameworks*. Porém, se importando apenas com casos gerais de uso, podem não ser eficientes na distribuição de carga e recursos. Os trabalhos mais recentes tem focado na arquitetura, e em alguns destes apresentando um módulo despachante de tarefas.

Liquid é um modelo dividido em duas camadas, de processamento e de mensagem. A camada de processamento é responsável por tarefas ETL, guarantindo os serviços através do isolamento de recursos. A camada de mensagem é baseada no modelo *publisher/subscribe*, que é utilizado como *broker service* para armazenar os dados e gerenciar a carga de trabalho (*dispatcher*).

O gerenciamento consiste em utilizar *round-robin fashion* ou uma função *hash* para cada *producer partition*. Os consumidores são divididos em grupos, assim flexibilizando como os clientes consumem os dados.

A avaliação deste modelo foi feita através do sistema do LinkedIn com cinco *datacenters* distribuídos. As métricas coletadas apenas indicavam que é essencial reagir aos problemas rapidamente.

JetStream propõe diversas estratégias para transferência de eventos entre clouds. O JetStream *dispatcher* tem como objetivo aumentar a performance de transferência criando pequenos pacotes de eventos e podendo se adaptar às condições do fluxo de dados montiorando uma série de parâmetros.

A Arquitetura SMART considera sistemas heterogêneos distribuídos considerando custo, tolerância a falhas, *overhead* de rede e minimizar as transferências entre os nós de processamento.

A arquitetura consite em cinco grandes módulos, *Central Monitoring* possibilita visualizar e monitorar as condições de *clouds*, *grids* e dispostivos IoT. O *Global Aggregator* agrega os resultados e os mantém armazenados para o usuário final; *Core Engine*, por sua vez, mantém aplicações Big Data de *Batch* e/ou *Stream* através de *Clouds*, *Grids* e etc. O *Storage* possibilita armazenar os dados brutos e *Global Dispatcher* tem como objetivo desacoplar os dados coletados utilizando *message queues*. Nos experimentos realizados por Julio César demonstraram que o tamanho das entradas afeta a execução do Kafka e Flink, onde a performance melhora com blocos de dados maiores.

O Neptune é uma solução com foco na otimização dos recursos computacionais utilizados. Os orquestradores, também chamados de grânulos, disparam as tarefas para cada nó, que agem como containers. As tarefas são executadas utilizando cronogramas que podem ser direcionadas aos dados, periódicas ou por contagem.

Como visto acima, diversos modelos foram implementados para melhorar a performance da infraestrutura de processamento Big Data. Entre as estratégias adotadas podemos notar duas abordagens, uma sobre otimizar a comunicação e outra sobre o particionamento de *streams*.

B.4 Aten: modelo proposto

O Aten, é um *dispatcher* para sistemas Big Data que utilizam tanto *Batch* quanto *Stream processing* em ambientes heterogêneos. Sua construção utiliza diversas tecnologias *open-source* assim como algoritmos presentes no estado da arte, com o objetivo de solucionar problemas comuns, porém complexos desses sistemas.

O *dispatcher* é constituído pelo *Ingestion module*, responsável por reunir e manipular a carga de trabalho e o *Controller module* gerencia essa carga, conforme a Figura 5.1. Além disso, existe uma camada chamada de *resource watcher*, que possui uma visal geral do *dispatcher* e parcial do sistema, para ajudar no gerenciamento de recursos dentro dos componentes da arquitetura.

A conexão entre os clientes que produzem dados e a fila de mensagem fica a encargo do *Ingestion module*. Os clientes se comunicam com o sistema via *requests*, então o *Ingestion module* regula a relação com o sistema de *batch* e *stream*, separando os *requests* e os encaminhada a fila adequada. A implementação desse módulo utiliza um algoritmo de *buffering* que agrega os eventos de stream, ademais acumula os eventos que estão em *buffer* com a intenção de manter certo nível de tolerância a falhas.

Inicialmente, ocorre a identificação do evento como *batch* ou *stream*, então o evento é acrescentado ao seu devido *buffer* e quando este atinge o tamanho máximo envia uma única vez todos os eventos agregados. O tamanho do *buffer* traz certas considerações, visto que o compromisso com a velocidade dos resultados geralmente variam de aplicação para aplicação e achar seu tamanho ideal não é uma tarefa trivial. Visto isso, o Aten fornece opções em como agregas os eventos: por tempo e/ou número de eventos.

O *Controller module* contém as *message queues*, como Apache Kafka e RabbitMQ, o *event manager*, *partition balancer* e o DFS. O *event manager* identifica os eventos e o *batch* é enviado ao DFS, fazendo com que a aplicaç ao processe os dados quando estiverem num ambiente compartilhado. Se os eventos forem *streams* são encaminhados para o *partition balancer*, que por sua vez aplica algoritmos de repartição e conecta com o sistema de processamento levando em conta o paralelismo dos operadores, tipo de carga ou usando outro algoritmo.

Há diversos algoritmos de particionamento, *naive*, *rescale*, *broadcast*, *random*, *rebalance*, *round-robin*, sendo o último o mais comum. O algoritmo *naive* distribui as mensagens proporcionalmente ao *throughput* dos nós de processamento, já o *rescale* utiliza o nível de paralelismo. O *broadcast* distribui os eventos para todos os recursos do cluster, *random* seleciona o destino das mensagens randomicamente. Já os *rescale* e *rebalance* tem sua implementação baseada em (KATSIFODIMOS; SCHELTER, 2016).

B.5 Avaliação e resultados

A plataforma de nuvem Azure foi utilizada como infra-estrutura para os experimentos do trabalho. Essa decisão de plataforma foi tomada devido o uso constante em avaliações de modulos e arquiteturas de big data na nuvem pública e privada. A arquitetura de experimentação seguiu a proposta da arquitetura Lambda e teve seus componentes distribuídos em diferentes *data centers*.

O cluster de execução é composto por máquinas virtuais complemente heterogêneas e esta descrito na tabela 6.1. O ambiente éseparado em três locais distintos: um produz os dados a serem processados; o segundo armazena os dados em fila de mensagens; e o terceiro consume os dados.

A metodologia de experimentação segue o modelo proposto por (JAIN, 1991), assim designs de experimentos são desenvolvidos considerando as seguintes variáveis de configuração do *dispatcher*: a primeira variável é definida por um de cinco algoritmos de particionamento de dados juntamente com o fluxo padrão, que não possui algoritmo de particionamento; a segunda é a presença da técnica de otimização de transmissão, utilizando um buffer na transferência das mensagens, e não o utilizando; e por fim os mesmos experimentos são replicados 30 vezes, buscando dispor de uma amostragem mais completa dos resultados. Todos os experimentos são executados em sequência, porém o ambiente é isolado o máximo possível, reiniciando assim dados de cache, as configuraçõe e por fim os*frameworks*, além disso os testes em sequencia não são repetidos.

B.6 Conclusões

Aten é um modelo de alto nível, que pode ser utilizado entre as filas de mensagens e o motor de processamento híbrido, que busca otimizar o uso da comunicação e a distribuíção do fluxo de mensagens em ambientes heterogêneos, culminando o uso da rede, CPU e memória.

Os resultados demonstraram a efetividade do uso de diferentes algoritmos no Aten, permitindo a ampliação de *throughput* e uso de recursos. O algoritmo rescale apresentou a maior efetividade para todos os cenários, contudo é possível que esse resultado seja diferente em aplicações ou infraestruturas diferentes.

É possível perceber também que mesmo para os algortimos que apresentaram resultados inferiores, como o *Random* e *Rebalance*, foi possível aumentar o *throughput* médio da aplicação utilizando o modulo de *buffering* proposto no modelo sobre diferentes tamanhos de agregação. No entanto, buscando ampliar ainda mais o *throughput* de sistemas de fluxo de dados é necessário uma avaliação mais profunda, visto que a performance da aplicação depende intrinsecamente da carga de trabalho e a disponibilidade de recursos.

Aten foi modelado com a premissa de um modelo genérico que permite a manipulação de seus componentes a fim de adaptar-se para diferentes arquiteturas em diferentes ambientes. O modelo por si só, permite sua configuração para utilização de algoritmos e módulos ou a substituição de componentes disponibilizando assim, adaptabilidade do modelo.

Para os trabalhos futuros, pode-se avaliar o uso de agregação dinâmica, que se adapta ao comportamento das aplicações, usuários ou mudanças de ambientes voláteis. A definição de agregação é totalmente depende dos critérios definidos pela aplicação e seu ambiente, o algoritmo de particionamento sofre diretamente com a decisão de agregação do módulo inferior do *dispatcher*, o mesmo deve ser analisado para o desenvolvimento ou escolha de algoritmo de particionamento. Dessa forma, há espaço para diferentes propostas que visam solucionar diferentes problemas encontrados em múltiplos ou em cenários especificos.